

ADDC-TR-88-133, Vol III (of three)  
Final Technical Report  
June 1988



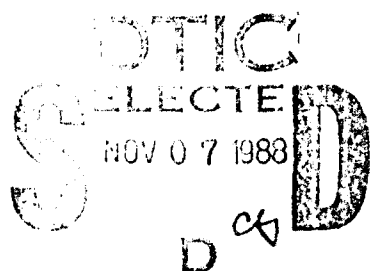
AD-A200 171

# THE C<sup>2</sup> SYSTEM INTERNET EXPERIMENT System/Subsystem Specification

BBN Laboratories Incorporated

James C. Berets, Ronald A. Mucci and Kenneth J. Schroder

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.



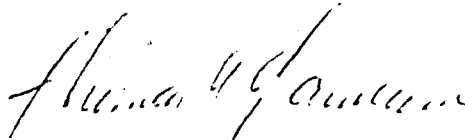
ROME AIR DEVELOPMENT CENTER  
Air Force Systems Command  
Griffiss AFB, NY 13441-5700

88 11 07 090

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

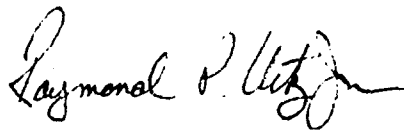
RADC-TR-88-133, Vol III (of three) has been reviewed and is approved for publication.

APPROVED:



THOMAS F. LAWRENCE  
Project Engineer

APPROVED:



RAYMOND P. URTZ, JR.  
Technical Director  
Directorate of Command & Control

FOR THE COMMANDER:



JOHN A. RITZ  
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notice on a specific document requires that it be returned.

UNCLASSIFIED  
SECURITY CLASSIFICATION OF THIS PAGE

HP 4200 171

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS N/A		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.		
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) 6248			5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-133, Vol III (of three)		
6a. NAME OF PERFORMING ORGANIZATION BBN Laboratories Incorporated		6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTD)		
6c. ADDRESS (City, State, and ZIP Code) 10 Moulton Street Cambridge MA 02238			7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700		
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Rome Air Development Center		8b. OFFICE SYMBOL (If applicable) COTD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER P30602-84-C-0140		
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700			10. SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO. 62702F	PROJECT NO. 5581	TASK NO. 21
			WORK UNIT ACCESSION NO. 70		
11. TITLE (Include Security Classification) THE C <sup>2</sup> SYSTEM INTERNET EXPERIMENT - System/Subsystem Specification					
12. PERSONAL AUTHOR(S) James C. Berets, Ronald A. Mucci and Kenneth J. Schroder					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM Aug 84 TO Mar 86		14. DATE OF REPORT (Year, Month, Day) June 1988	
15. PAGE COUNT 122					
16. SUPPLEMENTARY NOTATION N/A					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	Distributed Operating System, System Monitoring and Interoperability, Control, c. Heterogeneous Distributed System, Survivable Application.		
12	07				
19. ABSTRACT (Continue on reverse if necessary and identify by block number) This is the Final Technical Report representing work performed for the C2 System Internet Experiment project. The C2 Internet Experiment project is a complementary effort to the CRONUS Distributed Operating System (DOS) project. Under the C2 Internet Experiment, we designed and built a prototype distributed application to demonstrate CRONUS concepts and to provide an environment for evaluating CRONUS' current suitability for supporting distributed applications, especially applications that pertain to the command and control environment. Both the C2 Internet Experiment and CRONUS DOS projects are part of a larger program being supported by the Rome Air Development Center (RADC) to design, develop and support an appropriate technology base for building the types of distributed systems which are expected to be fundamental in future command and control applications and elsewhere.					
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas R. Lawrence			22b. TELEPHONE (Include Area Code) (315) 330-2158		22c. OFFICE SYMBOL RADC (COTD)

## Table of Contents

<b>1. INTRODUCTION AND OVERVIEW</b>	<b>1</b>
1.1 Background	1
1.2 Project Objectives	3
1.3 Report Overview	4
<b>2. APPLICATION DESCRIPTION</b>	<b>5</b>
2.1 Application Architecture	5
2.2 The Application Scenario	8
2.2.1 Initial Scenario Configuration	8
2.2.2 Details of the Scenario	9
<b>3. THE CRONUS DISTRIBUTED OPERATING SYSTEM</b>	<b>13</b>
3.1 The Cronus System Environment	13
3.2 The Cronus Model	13
3.3 Cronus Core Functionality	19
3.4 Cronus Monitoring and Control	21
3.5 Cronus Support for Design	21
<b>4. SYSTEM DESIGN DETAILS</b>	<b>22</b>
4.1 Functional Decomposition	22
4.2 Design Details	23
4.2.1 Timer Manager	26
4.2.2 Target Simulation	28
4.2.3 Sensor Manager	29
4.2.4 Mission Data Manager	30
4.2.5 Fusion (Detection Correlation) Processing	31
4.2.6 Target Report Manager	32
4.2.7 Meteorological Data Manager	32
4.2.8 Cartographic Data Manager	33
4.2.9 Resources and Logistics Data Management System	34
4.3 Design Evolution	35
<b>5. SYSTEM IMPLEMENTATION</b>	<b>37</b>
5.1 Implementation Strategy	37
5.2 Component Implementations	38
5.2.1 Timer Manager	38
5.2.2 Target Simulation	42
5.2.3 Sensor Manager	43
5.2.4 Mission Data Manager	47
5.2.5 Fusion (Detection Correlation) Processing	50
5.2.6 Target Report Manager	51
5.2.7 Meteorological Data Manager	52
5.2.8 Cartographic Data Manager	53

5.2.9	Resources and Logistics Data Management System	56
5.2.10	Application Monitoring and Control	56
6.	SUMMARY AND CONCLUSIONS	61
7.	REFERENCES	62
A.	APPENDIX: APPLICATION COMPONENT DOCUMENTATION	63

## FIGURES

Functional Overview of C <sup>2</sup> System	6
Detailed Functional Description of Application	7
Sensor Observing Ground Targets	10
Object Model in General, Cronus, and C <sup>2</sup> Internet	15
Client/Manager Communication in a Multihost/Multicluster Configuration	16
An Example of Client/Manager Communication in C <sup>2</sup> Internet	17
BBN Cluster Configuration showing Cronus System Services	20
Multicluster Assignment of C <sup>2</sup> Internet Functional Elements	24
Overview of Experimental C <sup>2</sup> Internet System Currently Designed	25
Overview of Experimental C <sup>2</sup> Internet System Currently Developed	39
Sensor Manager's Operatesensor Operation.	45
Example Target Trajectories	49
Cartographic Display Image with Target Tracks	57
Application MCS Host Status Display	59
Application MCS Service Status Display	60



Addendum For	
100- GRA&I	<input checked="" type="checkbox"/>
100- TAB	<input type="checkbox"/>
100- TAB	<input type="checkbox"/>
A-1	

## TABLES

Timer Object Attributes	41
Target Object Attributes	43
SEGMENT Attributes	43
Sensor Object Attributes	46
FRMDEF Attributes	46
Mission Schedule for Radar Sensor <i>neurad</i>	47
Mission Schedule for Infrared Sensor <i>ifs</i>	48
Mission Data Attributes	50
DETECTIONFRAME Attributes	50
Target Report Attributes	51
TARGETDESC Attributes	51
SENSORDATA Attributes	52
DETECTIONS Attributes	52
Weather Report Object Attributes	53
Weather Forecast Object Attributes	54
Feature Map Object Attributes	55
Vehicle Object Attributes	56

## 1. INTRODUCTION AND OVERVIEW

This report presents specifications for the system and subsystem components of the C<sup>2</sup> Internet Experiment application. C<sup>2</sup> Internet is a complementary effort to the Cronus distributed operating system project; both C<sup>2</sup> Internet and Cronus are part of a larger program to develop and support an appropriate technology base for building the types of distributed systems which are expected to be fundamental in future command and control systems. The Cronus system development project is an ongoing project which is developing an architecture, an operating system, and software development tools to support distributed command and control applications. Under the C<sup>2</sup> Internet Experiment we are designing and building prototype applications to demonstrate Cronus concepts and to evaluate Cronus' current suitability for supporting command and control applications.

This report focusses on specifications for various components and how those components are designed and built. This report represents a continuation of the design initially presented in C<sup>2</sup> Internet Experiment: Interim Technical Report [BBN-6073], and includes details of components which have been designed after that report was published. We assume that the reader is already familiar with the functional description of the application presented in [BBN-5942].

### 1.1. Background

It is inevitable that many military systems of the future will be both automated and physically distributed. These systems are well-matched to the potential benefits of a distributed architecture. Collections of these systems are necessarily larger and more complex than their component systems. With current technology, there is no alternative to a distributed architecture for such cooperating systems. Many systems obtain survivability through physical dispersal and redundancy [ADDCOMPE83, TRW80]. In these cases too, a distributed system architecture is essential.

Once we have accepted the idea that future systems will be distributed, there are a number of other desirable characteristics which the support architecture should exhibit. Functional specialization generally necessitates that different types of computers will be available and used to solve different parts of an overall problem, yet the result must continue to perform as an integrated system. System distribution should not diminish the usual requirements for effectively controlling the management of system resources or the desire to control access to these resources, although it does render ineffective current mechanisms in these areas which are limited to a single computer system. Experience has shown that large complex systems cannot be conceived whole and remain static, but rather need to evolve with time, changing requirements, and technology. No matter what support components we choose for an implementation, there will inevitably be frequent growth and changes which need to be anticipated. A distributed system architecture typically contains many components. This results in a situation of constant change due to the number of components involved. Taken together, there are many formidable barriers toward achieving the sought after benefits of a distributed system architecture.

A system development project incorporating the above requirements using available off-the-shelf technology would currently have available only a collection of independent host computers, some communication medium, and some standard software communication protocol support for transporting uninterpreted data between hosts. The gap between the problem-oriented requirements and the system-

oriented support is huge. As a consequence, the application designer is forced to deal directly with a variety of support areas to fill this void. This added burden will at best make such projects much more complex and expensive, and at worst cause many of them to fail.

One approach to changing this situation is through the development of a distributed operating system (DOS) and its support functions. A distributed operating system provides computing and communication resources to application programs, while promoting resource sharing among interconnected computer systems, and managing the collection of resources that are shared. A DOS bridges the gap between applications and communication by providing a coherent and integrated system for supporting the development and operation of distributed applications.

Recognizing the existence of such a gap, the Rome Air Development Center (RADC) has for a number of years been supporting research into the design and implementation of distributed operating systems. In 1981 the design of the Cronus distributed operating system was begun as an attempt to capture and extend much of what had been learned from previous research, including early investigations under the sponsorship of the Defense Advanced Research Projects Agency (DARPA). The intent was to provide a testbed for application developers to gain hands-on experience with designing, building, and evaluating distributed applications. The Cronus testbed provides application developers with exposure to Cronus concepts and software for developing distributed applications, and direct experience with the effects of distribution on their application.

As a means of testing and evaluating the implementation of Cronus to date (see [Schantz86, Gurwitz86, BBN-5879; and BBN-5884]) we are now developing an exemplary application chosen from the command and control area. This testbed approach to demonstrating the utility of the Cronus tools and concepts for developing distributed applications is referred to as the C<sup>2</sup> Internet Experiment. The application area is a set of battle management functions which encompass the various aspects of distributed systems. The emphasis in the application development at this time is on the interfaces between components that comprise the application, and the use of distributed resources by these components. It is not on the fidelity of the command and control system components implemented.

There are a variety of uses of Cronus to be explored in the experimental application. One important role of the DOS is to support and control interoperability between resources available on different host systems. A second role is to facilitate the development of a multi-host implementation of an integrated function or subsystem (e.g., one of the application resources) as an alternative to a single host implementation, to achieve survivability, scalability, and other desirable attributes. A third role is to facilitate the gradual evolution from existing but largely non-integrated computing resources towards an effective integration and management of resources and services better able to capitalize from interconnection. Each of these uses of Cronus are discussed further within the context of the C<sup>2</sup> experiment.

## 1.2. Project Objectives

There are a number of objectives being simultaneously pursued in the C<sup>2</sup> Internet Experiment, among them the following.

- We are attempting to demonstrate and evaluate the Cronus concepts and tools in developing a prototype application with requirements similar to projected uses.
- We are using the existing Cronus testbed hardware/software implementation to evaluate its performance and reliability characteristics under conditions approximating intended application uses.
- We are using the experience gained from matching application requirements to Cronus support mechanisms, and from developing the prototype application, to help guide the direction of future system enhancement, extension and repair, and to further refine the methodology used for migrating from application definition to running system.
- We are using the application domain to focus appropriate problem-specific support for various system attributes such as survivability, global resource management, and multi-cluster operation, as well as to provide a means for the hands-on evaluation of an initial implementation of these properties within the context of the application.

One of the first tasks accomplished under this effort was the selection of an appropriate application or set of applications. The variety of areas in which distributed system approaches apply has had an impact on application selection. If the application was too limited, we would surely miss important areas for both demonstration and evaluation; if it was too large and dependent on details, we could never make any initial progress under limited budget constraints. It is desirable to have an application domain which includes a rich set of distributed system problems, can be adequately demonstrated without excessive attention to details of the application, and is extensible to allow incorporation of improvements in the technology and additional effort in application development. To make test and evaluation more meaningful, we selected a suite of applications which are immediately identifiable as close to possible intended usage patterns; we hope that the application-independent nature of much of the underlying system support is clearly recognizable. The application was described in detail in the Functional Description report (BBN-5942). We have subsequently established prototype scenarios of use within this application domain for demonstration and evaluation purposes.

As an initial step in the application design process, we have decomposed the application along functional lines in keeping with the Cronus application development paradigm, and established preliminary interactions among functions to support the selected scenarios. We have done a detailed design for distributed versions of a few of these application functions and client programs to exercise these functions. The functions designed to date include sensor simulation, sensor data collection and display, support functions for meteorological and cartographic services, support for a limited C<sup>2</sup> resource data base, application monitoring functions, and support for initiating and controlling experiments.

We have implemented initial versions of each of the functions which have been designed to date, using the Cronus development tools and other Cronus support mechanisms. We have intentionally designed and implemented some subsystems before even designing other subsystems both to achieve an early demonstration capability and to reinforce the utility of an evolutionary approach to system development. Similarly, the components developed to date represent only a preliminary implementation

of the full functionality appropriate to the application, emphasizing those aspects which focus on distributed system technology. These functions will themselves evolve with time. In the course of developing these initial system components to run under Cronus, and in using the Cronus application development tools, there have been numerous enhancements and refinements to Cronus, and a number of other areas have been targeted for future development.

These design and implementation activities have now been integrated into the first significant C<sup>2</sup> Internet application demonstration capability.

### 1.3. Report Overview

The remainder of this report is divided into five sections and an appendix. Section 2 briefly reviews the application description and functional components of the C<sup>2</sup> Internet Experiment application; a more complete discussion can be found in [BBN-5942]. Section 3 focuses on the use of Cronus supplied system services and tools in the development of the C<sup>2</sup> Internet Experiment application. Sections 4 and 5 each focus on a particular phase of the specification and implementation process. The final section briefly reviews our experience. A more detailed review of the project can be found in the C<sup>2</sup> Internet Experiment: Final Report [BBN-6251]. The appendix presents detailed specifications for the application managers that have been built to manage resources used by the application.

## 2. APPLICATION DESCRIPTION

Of primary importance in choosing a  $C^2$  Internet application is that the application be a non-trivial example in the command and control area requiring a distributed heterogeneous architecture. To make the test and evaluation of Cronus meaningful, it is important that the application be immediately recognizable as pertinent to planned or desired usage patterns, and that the system achieve satisfactory interoperability in those activities performed during the experiment scenario.

The development of the application builds upon Cronus both in approach and in the use of underlying support. The object-oriented development approach that was used to develop the Cronus distributed operating system, has also provided the methodology for developing the  $C^2$  Internet architecture and components: the existing Cronus system functions, as well as Cronus support for distributed software development, implementation, and maintenance are being used to support the development of the experimental application.

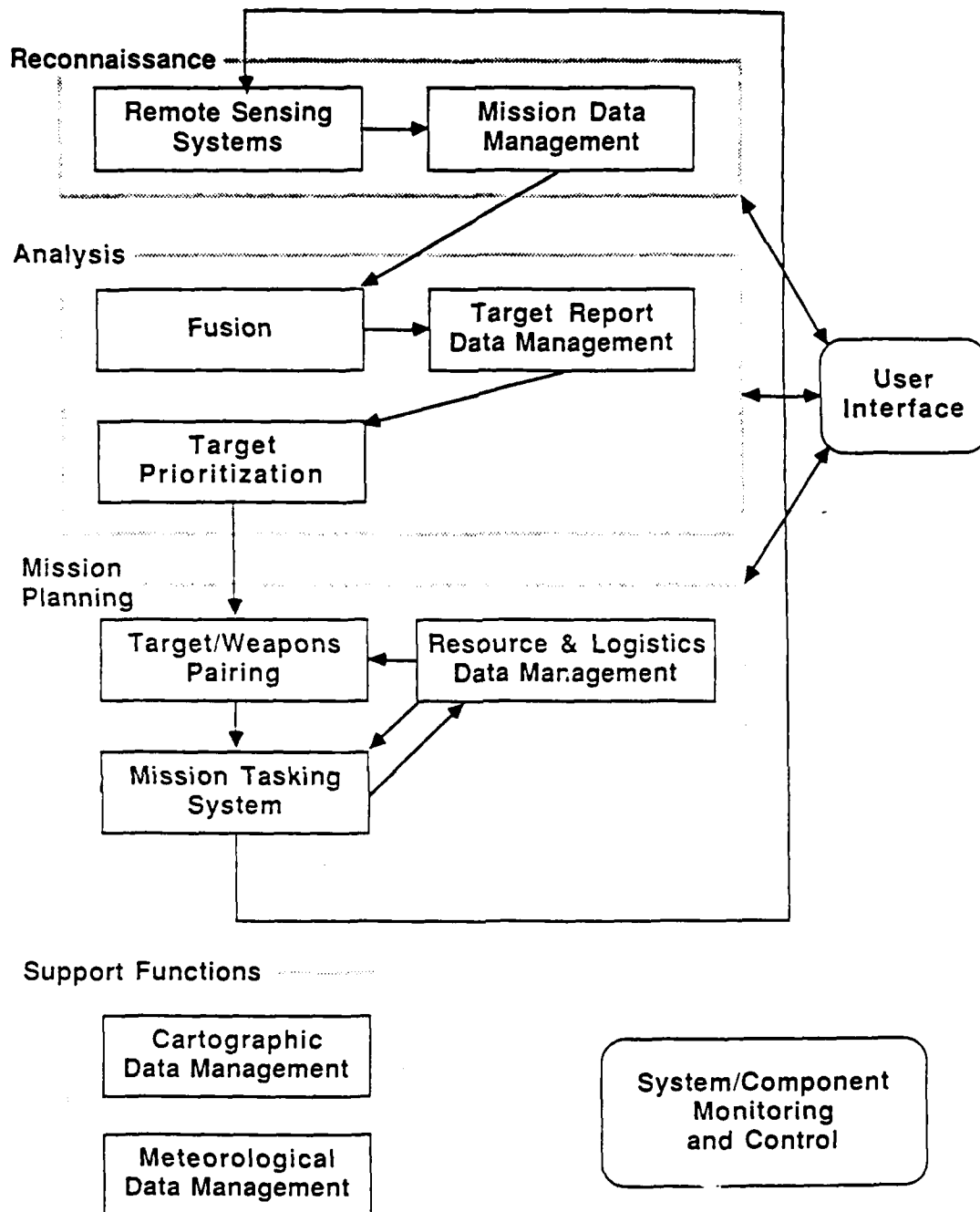
For us to make an adequate appraisal of the value of Cronus in developing distributed applications, an application is desirable that includes a rich set of distributed system problems, yet can be adequately demonstrated without excessive attention to details. The application should also be easily extensible in order to allow it to follow improvements in technology and the future development of underlying Cronus system support. This will facilitate evolution of the evaluation environment in concert with the evolution of Cronus. In the remainder of this section, we will summarize the  $C^2$  Internet application detailed in the Functional Description document [BBN-5942] and describe a typical scenario which will be used to exercise the application.

### 2.1. Application Architecture

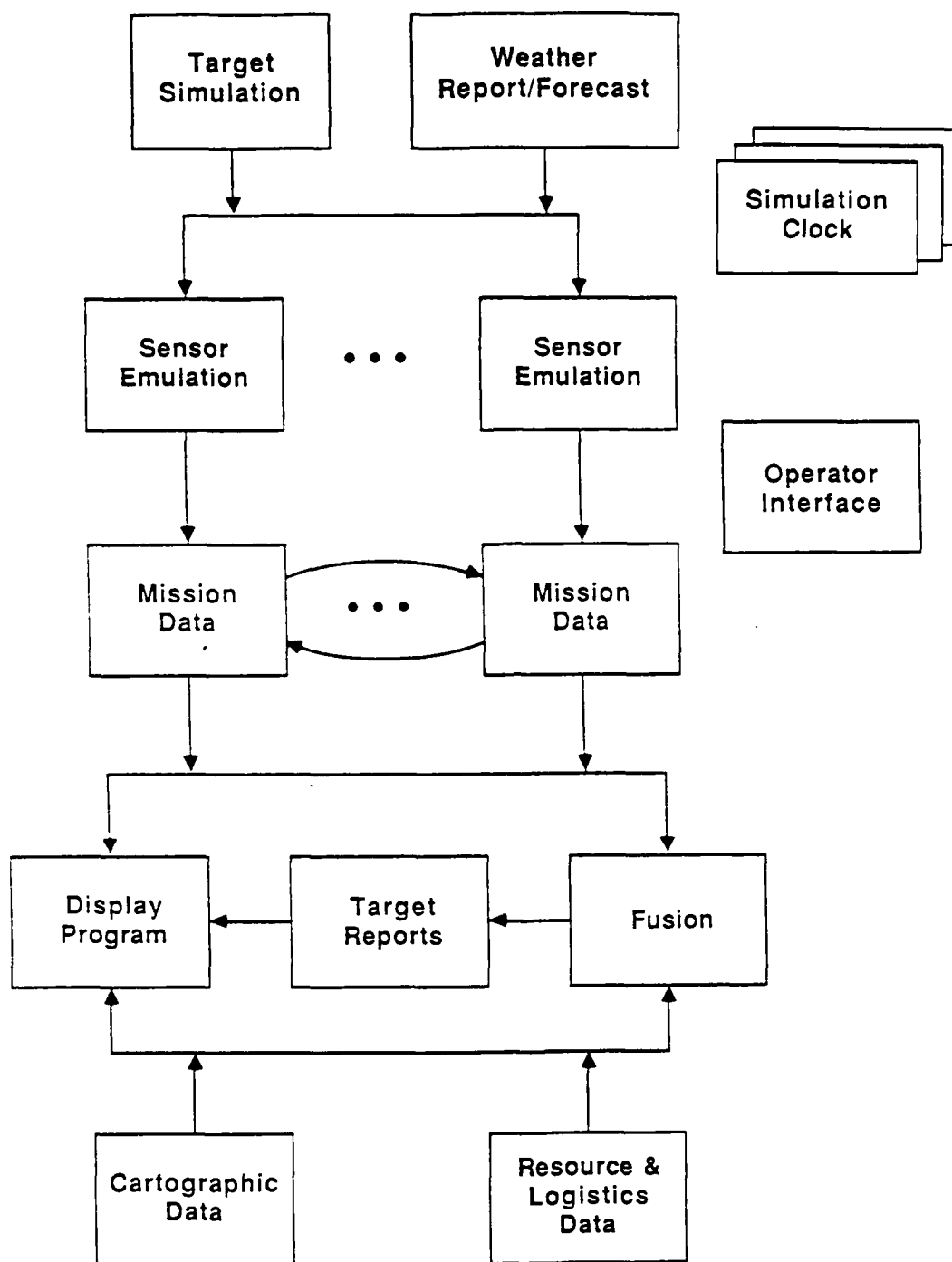
The overall architecture of the  $C^2$  Internet Experiment application reflects a process which recurs in many diverse  $C^2$  applications, namely repeatedly performing the following tasks:

1. *Collect data* from several sources, recording it when appropriate;
2. *Process the data* to assess a given situation, either by computer or an operator;
3. *Take action*, either to gain additional information, dispatch forces to intercept an intruder, or otherwise react to the data received; and
4. *Verify* the effects of the actions.

The application architecture selected for the  $C^2$  Internet Experiment is shown in Figure 1, and in more detail in Figure 2. It represents a collection of interconnected sources of data and processing elements which are the relevant constituent parts of a hypothetical tactical air control system for planning and supervising attacks against selected ground targets. Target data obtained from simulated remote sensing systems is the primary source of input for data fusion (target data correlation) and other processing. The target data correlator produces target reports for subsequent target situation analysis and assessment. The target track reports are the primary input to mission support operations such as target track prediction, target classification, identification, and prioritization, target/weapons pairing, and



Functional Overview of C<sup>2</sup> System  
Figure 1



Detailed Functional Description of Application  
Figure 2

mission tasking. Other data sources, including cartographic, meteorological, and logistics and support information, are also planned. Functions are also provided to support the monitoring and control of both experiment and application resources.

Some of these elements will be simulated; however, the process of application design, implementation, and user interface development mirrors the construction of an actual command and control system using Cronus tools and concepts. A detailed functional description of these components can be found in [BBN-5942].

## 2.2. The Application Scenario

The architecture of the C<sup>2</sup> Internet Experiment is, by design, capable of accommodating a wide variety of command and control elements. In order to clarify the relationship between the application architecture, its use in a C<sup>2</sup> situation, and the underlying distributed system base, we will outline an application scenario within which we expect to exercise the C<sup>2</sup> application and (indirectly) Cronus. This specific scenario will also provide the framework for a demonstration of Cronus within a specific command and control context.

### 2.2.1. Initial Scenario Configuration

The application will run on a heterogeneous collection of computers, eventually spread across at least two physically dispersed areas. Application functions will be assigned to these computers in a manner that exploits their specialized features. The assignment reflects the way in which functions might be assigned in a real system, where aspects of the computing environment may constrain component placement. Many of the functions provided are supported by a collection of several cooperating elements. This redundancy can be used to provide survivability of the services and the data they manage. It also allows duplicate resources to be placed near computers that use those services, potentially improving performance by reducing the delays that arise from communication and competition for the resource.

We will create several functional elements to participate in this simulation. The list we present here is just an initial one needed to build the backbone of the scenario. The underlying architecture allows this scenario to be extended by adding such detail as maps for new geographic regions, additional sensor systems to track vehicles, additional targets, etc., as well as by adding new functional elements to augment the scenario.

The setting for the scenario is a geographic region, currently 100km square, for which actual cartographic data is available. Within this region, a number of clusters of enemy vehicles will enter, be detected, tracked and identified as they move along roads. Their target, currently a power plant, will be predicted and interceptors will be dispatched to attempt to stop the vehicles before the power plant is destroyed. The progress of the experiment will be displayed from several perspectives, with cartographic information used as a background for regional displays.

Each cluster of vehicles will consist of several individual vehicles on similarly scheduled routes; the routes for the clusters will differ but initially have the same destination. These simulated targets and their routes are managed by a target simulator function

Initially, a combination of radar and infrared airborne sensing systems will be used for target detection. These sensors are deployed on aircraft, but can be controlled remotely from the command center through a ground station to which they relay their information. The sensors will be capable of sensing target results for only a portion of the region where the scenario takes place (Figure 3). Computers representing each ground station will simulate the entire sensor configuration. The processing of the sensor data will be done by ground-based, special purpose, *sensor data fusion processing equipment*.

The experiment will be performed under various meteorological conditions ranging from very clear to heavy rain or snow. This will affect the sensor performance, in terms of the accuracy of the target data the sensors report, and will be a consideration in the use of the available sensor resources, target/weapons pairing, and mission planning operations. The meteorological service will be accessed for all of these planned uses of weather information.

The target detections produced by the processing will be forwarded to operators in the command center for review. We will provide the capability of displaying the sensed targets, both as detected by the initial sensor processing, and as refined by the fusion processing. These operators will be equipped with their own computers, their exact number scaling to meet the number of expected targets.

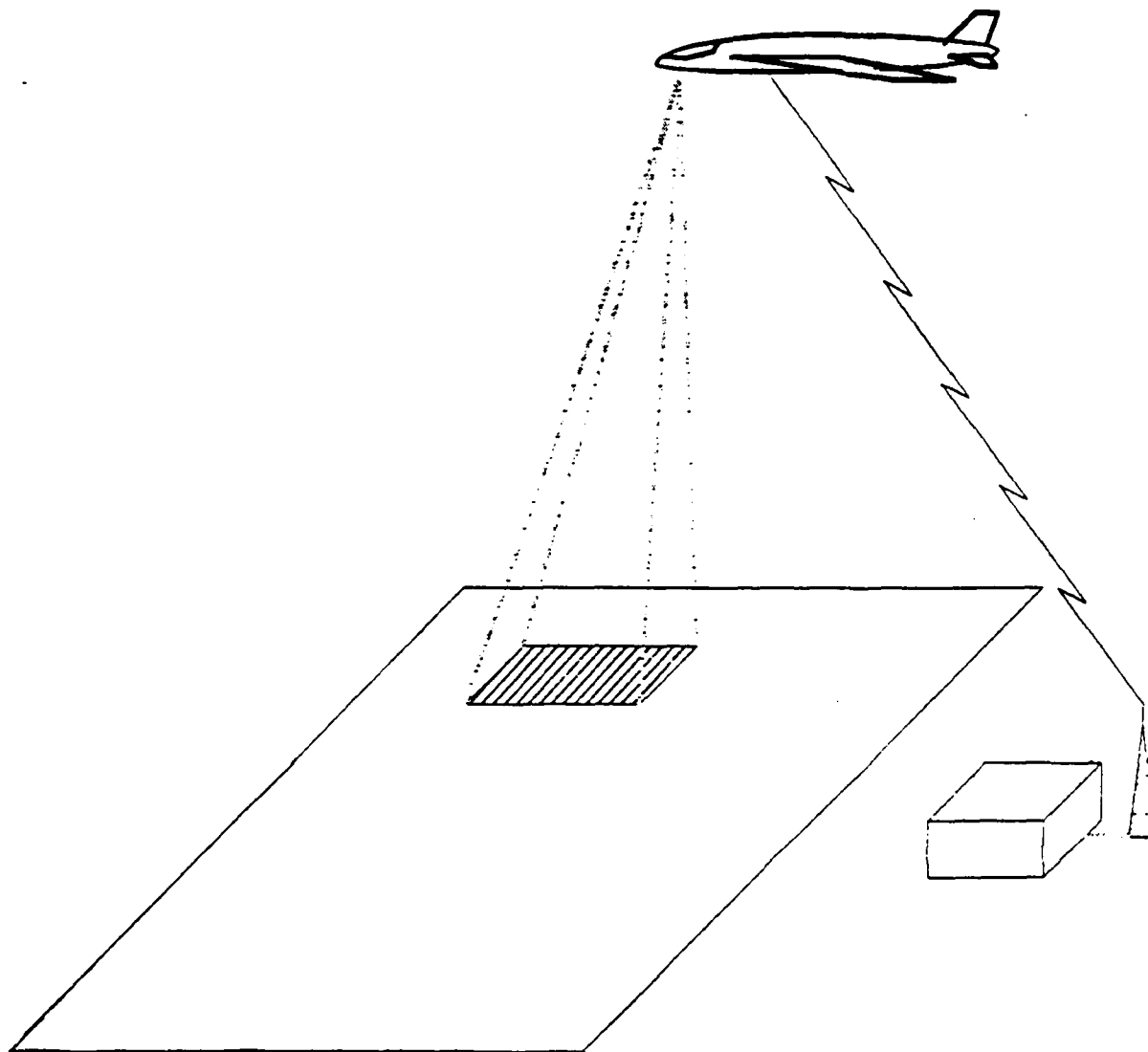
The Resources and Logistics Data Management System will include a limited set of data in the initial scenario. It will contain basic aircraft capabilities for different types of aircraft, airfield capabilities, such as runway length, and aircraft availability at the airfields.

### 2.2.2. Details of the Scenario

The sensor and battle management functions provide numerous opportunities for demonstrating, exercising, and evaluating distributed system architecture and Cronus support functions for interoperability, controlled remote access, survivability, and resource management in an application context. A detailed look at our initial scenario sequence will help to highlight these points.

As the scenario begins, sensors are deployed to various sections of the region. Two different types of sensors, one radar and one infrared, will provide overlapping coverage in a single section; the remaining sensors will observe other sections of the region. The output of each of the specific sensors may be examined by an operator from an appropriately configured access point.

Vehicles will be detected in the section being monitored by both radar and infrared sensors. Normally, these reports include a mixture of actual and false detections. The detection threshold of the sensors can be set to produce various experimental situations. Any operator with appropriate access control rights can adjust the threshold to regulate the fraction of false detections that can be expected. If the threshold is set high, possible targets may go unnoticed when they enter the observed area. If the



Sensor Observing Ground Targets  
Figure 3

threshold is set low, a large number of false detections may lead the operator to dismiss the reports<sup>1</sup>.

When a potential target is identified, effective interoperability can be demonstrated by a timely and successful response to the event. For example, a target's appearance might warrant the redistribution of sensor resources or the reconfiguration of processing, display, and human operator resources. In this scenario, the initial operator review of radar image data will show an inconclusive situation.

Combining data from multiple sensor types will give more accurate target detections. To realize the full benefit of the multisensor configuration, that data must be remotely accessible in a timely and reliable manner. The primary user of such distributed data is a Target Data Correlation system, which is responsible for managing target data for a particular geographic region. When the new targets are discovered, an appropriate Target Data Correlation station will be chosen to assess the situation and determine what action is appropriate. This is essentially a resource management issue, where the resources being managed are the processors, displays, and operators associated with the Target Data Correlation stations.

The discovery of a new target, either by automatic processing or by an operator's observations, will result in an *alert* event. The target alert events will be used to set the command and control part of the demonstration into action. The end result of the various sensing functions will be a list of identified targets consisting of target location, speed, direction, and other distinguishable features such as size or temperature.

Following a target's discovery, target prioritization can take place. Periodically, priorities are assigned to active target objects. In our current scenario, targets will be assigned absolute priorities on a periodic basis. That is, groups of targets will be classified as the need arises. The priority of a target could be updated a number of times, by operators performing functions anywhere in the configuration, as the battle situation changes.

For convenience, we choose a particular threshold priority. Targets above that priority threshold are important and need to be acted on. Targets below the threshold will be ignored, although they may be examined by an operator. Prioritization acts, in some sense, as a target filter, separating important and unimportant targets.

Prioritization is probably done by someone who has a relatively global view of many aspects of the entire system. In order to make well-informed prioritization decisions, various state information must be readily available: target positions, weather, cartographic information, etc. Thus an operator station, responsible for the prioritization function, must be provided with convenient access to and flexible displays of this information, accumulated from the various computers which support those functions.

C<sup>2</sup> resource capability data, such as aircraft range, will be the primary information source in the target/weapons pairing function. Target/weapons pairing is concerned with the capabilities of classes of resources, not the availability of particular assets. In addition to resource capability, other data such as meteorological data may be needed to make appropriate target/weapon decisions. This data is dynamically accessed as needed from the operator's station.

<sup>1</sup>By controlling the simulated weather, we can also show the effect of weather conditions on the target data reported by the various sensors.

Mission tasking is another focus in the command and control part of the demonstration. The mission tasking function assigns specific resources to designated targets. Thus it relies heavily on resource availability data. The Mission Tasking function will choose from the available resources, based on information pertinent to a number of aircraft and airfields, and assign them to targets. Combinations of available resources can be assigned to the targets in a variety of ways. Some of this expertise will be embodied in a program which serves as an aid in avoiding inappropriate choices (e.g., using a non-all-weather aircraft during a storm, or an aircraft based too far away from the target).

Throughout the scenario, selected failures will be induced, such as taking one or more computers off-line. This allows us to demonstrate and evaluate the survivability properties of both functional elements and the application-oriented procedures which comprise the  $C^2$  scenario. In addition, scenario-oriented loading factors, such as number of targets, will be varied to demonstrate and evaluate scalability and resource management features.

An application monitoring and control station will be used to initiate, evaluate, and control system behavior under the simulated conditions. Additional experiment control programs are used to construct and control the simulation. These comprise another functional element of the architecture; however, its role is not to participate directly in the simulation, but rather to exercise global control of the simulation environment. This function can be used by any suitably authorized user, from anywhere in the configuration, to start and stop the simulation, create or modify objects that participate in the simulation, change the simulated weather conditions, and perform other operations that affect the simulated environment under which the experiment takes place.

### 3. THE CRONUS DISTRIBUTED OPERATING SYSTEM

The Cronus distributed operating system is intended to promote resource sharing among interconnected computer systems, and manage the collection of resources which are shared. Its main purpose is to provide a coherent and integrated system to support the development and use of distributed applications. In this section, we will briefly review the Cronus environment, system architecture, and programming support tools and discuss how they relate to the development of an application such as C<sup>2</sup> Internet. The foundation developed here will be used as the basis for a detailed discussion of the design and implementation of the C<sup>2</sup> Internet application in the following sections.

#### 3.1. The Cronus System Environment

Cronus exists in an environment of processors (or *hosts*) interconnected by computer communication networks. A Cronus *cluster* consists of some number of logically (and usually physically) grouped and interconnected hosts. Intracuster communication typically is characterized by relatively high available bandwidth and relatively low delay. Clusters may be interconnected; intercluster communication usually exhibits lower available bandwidth and higher delay than intracuster communication.

In the development of Cronus, we have taken the view that processing elements are heterogeneous. This assumption rests on the premise of functional specialization. That is, certain processors and systems will be more capable of (or more tailored for) certain functions than others. It is, however, highly desirable for these functionally specialized units to interwork so that they can cooperatively perform some larger task. Often each of the processors has its own native or *constituent operating system* (COS; for example, DEC's VMS). Instead of a tight coupling to one COS, Cronus is typically made minimally reliant on COS services. As a result, Cronus is largely independent of both the processor hardware and the constituent operating system upon which it is implemented. This increases the portability of Cronus functions, while minimizing changes to the COS due to the installation of Cronus. In addition, application programs may make use of both Cronus- and COS- provided services, thereby increasing their overall flexibility.

Consideration has been given to integrating Cronus more tightly with the native operating system on various hosts for efficiency reasons; however at this time the costs of this approach seem to outweigh the short-term benefits. Development is also under way toward the use of Cronus as a base operating system, initially for selected special-purpose system components.

#### 3.2. The Cronus Model

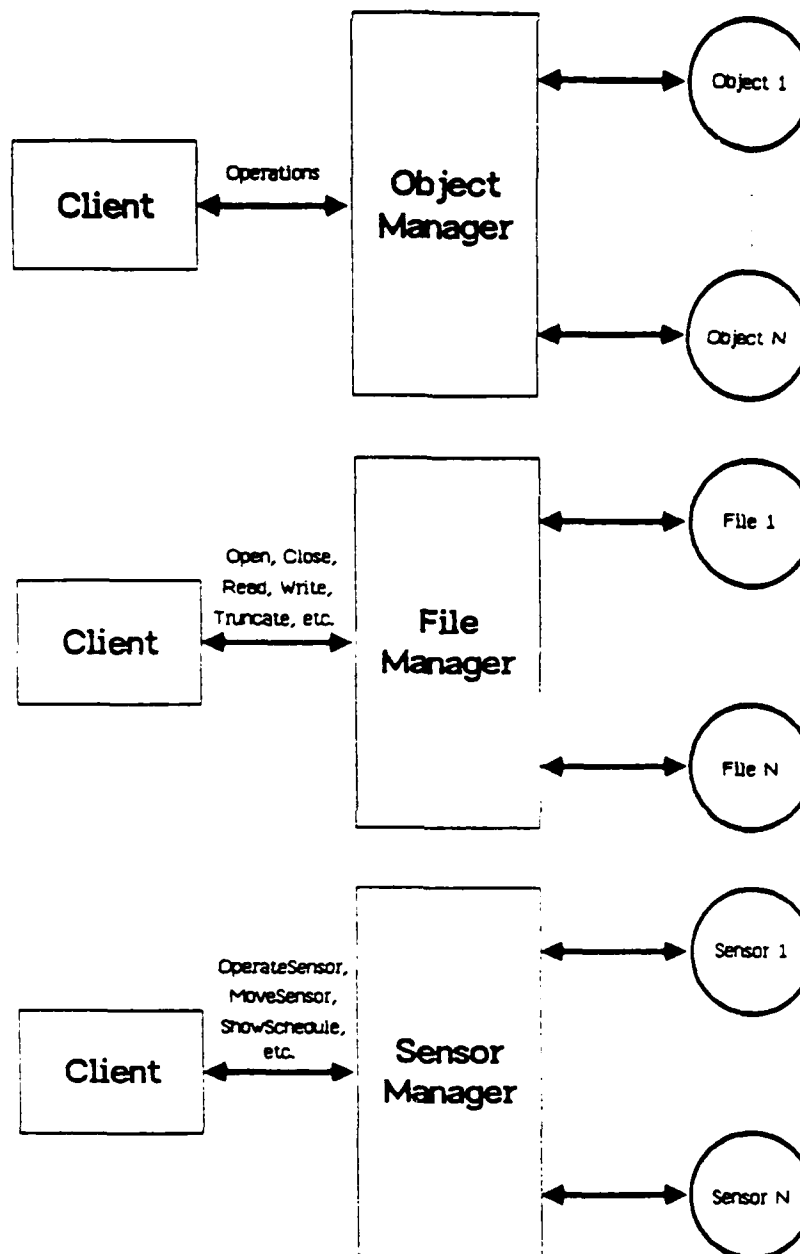
In order to provide the desired integrated view of diverse resources, a system model has been developed. The development of the model has been motivated by a number of factors including: the need for a coherent system decomposition, the need to provide easy to use "off-the-shelf" facilities to address distributed system problem areas for a wide range of potential applications, and the need to customize some of these facilities for various application components. A model which meets these needs is the *object model* on which Cronus is based. This model allows resources to be viewed abstractly,

shielding consumers of the resources from the actual implementation details that determine how and where the resources are provided. Using the object model, the distributed system is thought of as consisting of a collection of typed and uniquely-identified *objects*. Resources within the distributed system are viewed as objects, for example files, processes, and access control groups. Using its unique identifier, an object can be referenced uniformly from anywhere in the Cronus system. The object abstraction allows uniform system facilities to be used for a wide variety of interactions. An added benefit of the object model is that it is, by nature, easily extensible through the addition of new object types. Application resources are viewed as objects as well, and may make use of existing system mechanisms supporting the object model. Thus, in general (and more specifically within the context of the C<sup>2</sup> Internet application), part of the application design process is to define appropriate application objects and their attributes.

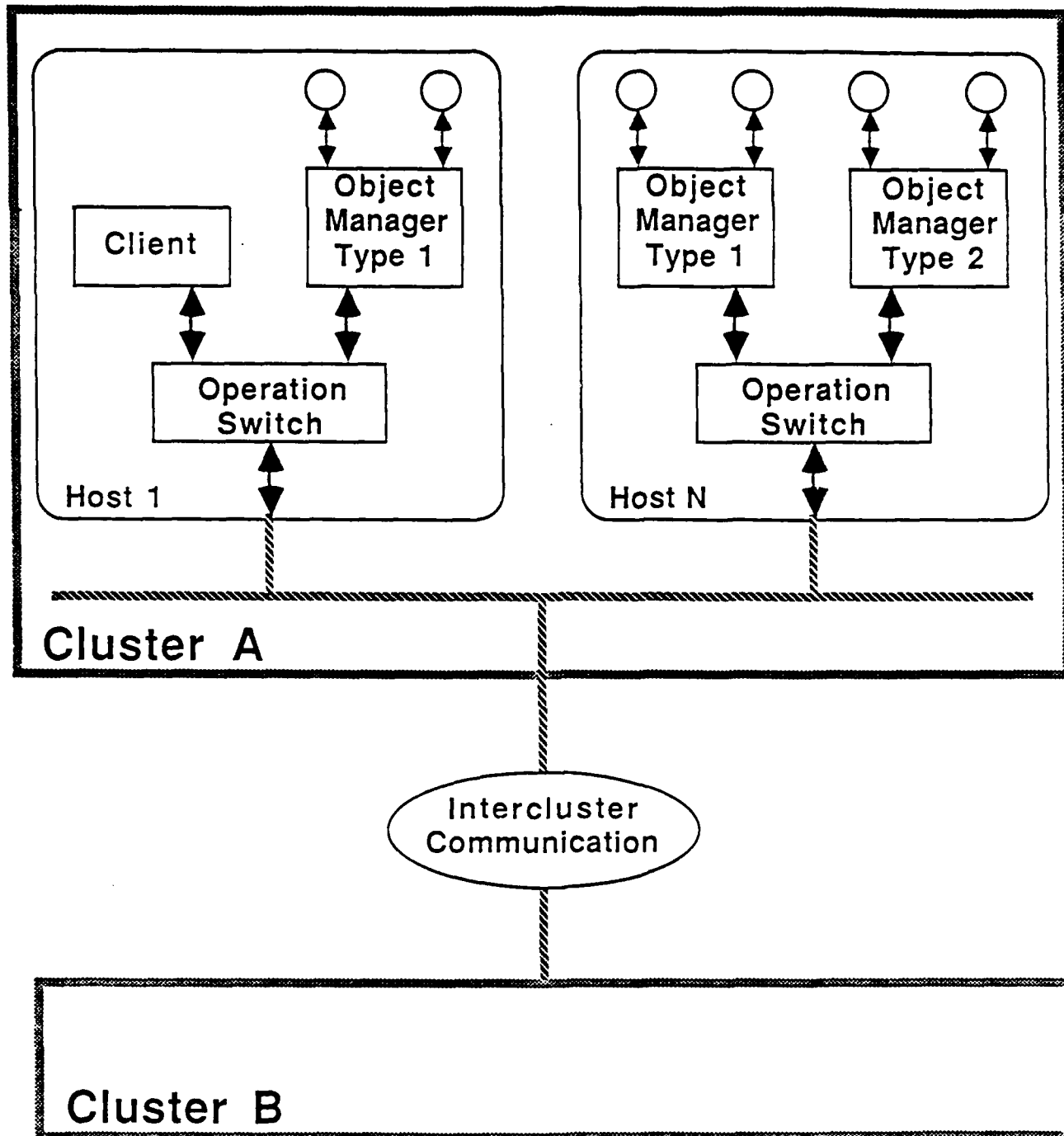
Associated with the objects of a particular type are a number of permissible *operations*. The operations on an object type are implemented within an *object manager*. An object manager is responsible for all objects of a particular type on a given host; one or more managers collectively manage a given object type within the Cronus environment. In order to perform tasks, clients (or other managers) invoke operations on objects. A client invokes an operation by sending a message to an appropriate manager over the network. These operations are then carried out by the object managers (see Figure 4). The Cronus system provides mechanisms for clients to manipulate objects without concern for their location in the network or for the implementation of the operations within the manager. This allows both transparent access to an object from anywhere in the network and the evolution of the mechanisms implementing various operations without necessitating changes to their external view.

Objects may be classified as *primal*, *replicated* or *migratory*. Primal objects are always located on the host on which they were created. Migratory objects may move from host to host as situations change. For example, when the storage resources on a given host are exhausted, a migratory file could be moved to another host to make more space available. Replicated objects are duplicated on one or more hosts and are used to enhance survivability. By having more than one active copy of an object, failures can be mitigated by using the redundant copy (or copies) of the object. The location of objects is, in general, transparent. That is, performing "local" and "remote" operations are functionally equivalent. However clients can control, if they wish, the placement of objects or the site for an operation invocation. The facilities for transparently locating and invoking operations on both primal and non-primal objects within the operating environment are supported by the Cronus kernel.

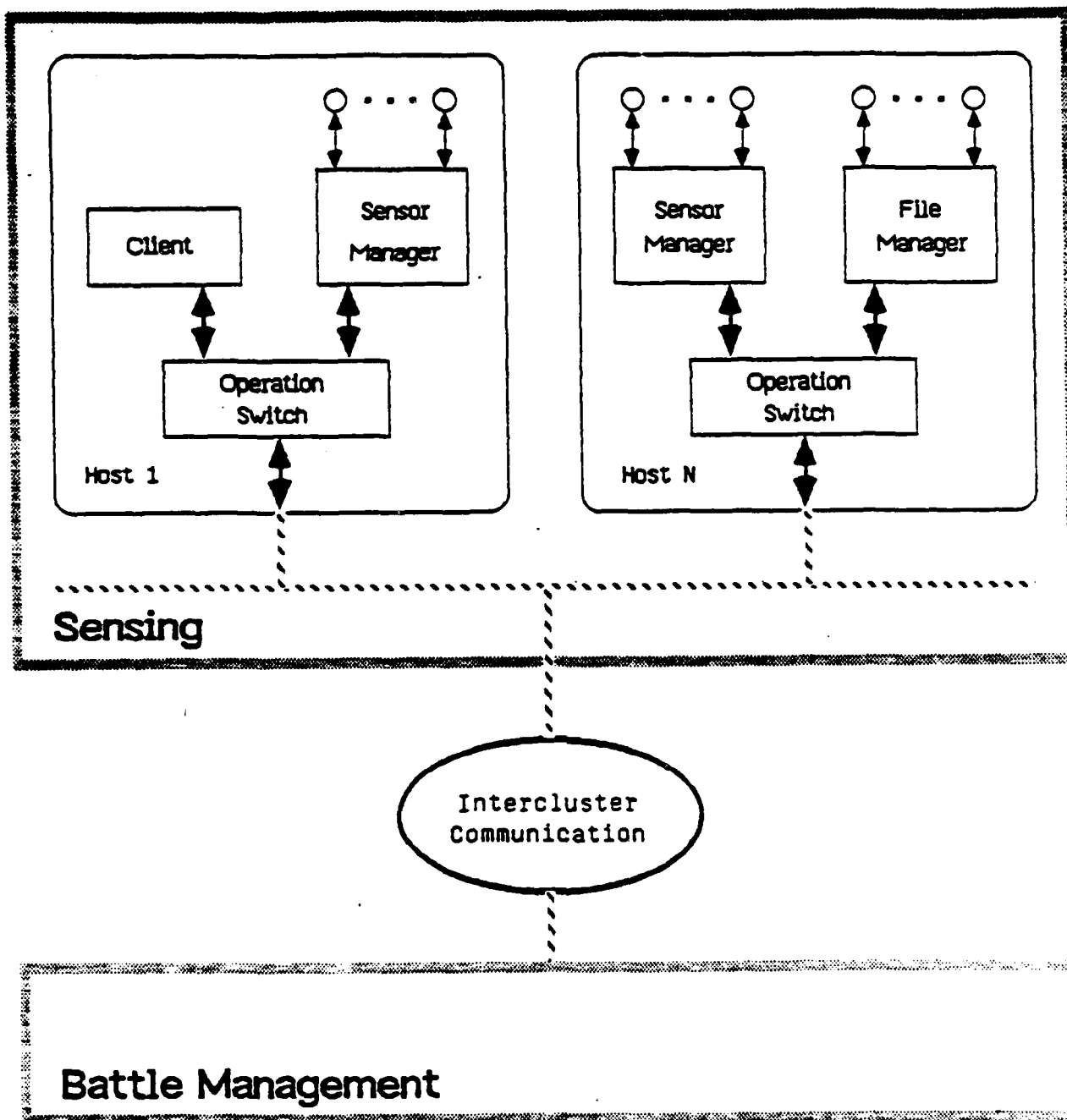
Clients and object managers communicate using an *interprocess communication* (IPC) mechanism contained within the *operation switch* which is part of the Cronus kernel [Schantz86]. The operation switch provides an object-based message passing facility. It implements or provides a supporting role for such functions as message routing, transparent location of migratory and replicated objects, and access control. The Cronus operation switch, in turn, relies on the datagram and virtual circuit services of the underlying communication network(s) to provide host-to-host communication. Figures 5 and 6 illustrate the structured communication and computation concepts of Cronus in general and for the C<sup>2</sup> Internet application respectively. Object managers communicate with both clients and other managers using standardized message formats and data representations. These *canonical* data representations allow convenient communication between heterogeneous hosts with differing internal host-specific representations. A set of built-in canonical types are contained within the basic Cronus system, such as the *EBOOL* (i.e., TRUE or FALSE), the *S16I* (a signed, sixteen-bit integer) and the *EDATE* (a canonical Cronus date and time representation); the set of data representations is easily extensible to include additional canonical types to suit particular applications. The use of canonical types in Cronus is



Object Model in General, Cronus, and C<sup>2</sup> Internet  
Figure 4



Client/Manager Communication in a Multihost/Multicluster Configuration  
Figure 5



An Example of Client/Manager Communication in C<sup>2</sup> Internet  
Figure 6

described in more detail in [Dean86].

This simple system model is used and extended to provide a number of important functions for distributed applications, including survivability, multi-host resource management, and system-wide access control. In the remainder of this section we briefly outline how these functions are provided within our system model in an application-independent manner.

Cronus provides facilities to make both its system functions and its applications survivable. Functional redundancy is offered by simultaneously providing more than one object manager (on different hosts) for a given object type. Data redundancy is provided by having multiple copies of objects. Managers of replicated objects cooperate to maintain the desired degree of consistency between the copies of the objects. This redundancy of both function and data is used to minimize the effects of failures and outages. Cronus authentication currently is implemented using replicated objects so that users may reliably be granted system access.

In a distributed system, the need arises for the formulation of global resource management policies, and for the mechanisms to implement these policies. (Resource management within specific hosts is generally the responsibility of the COS.) These strategies are necessary in order to effectively and conveniently control redundant resources. In Cronus, managers cooperate to enforce resource management policies. The Cronus resource management model is based on providing a set of mechanisms capable of supporting a variety of policies. These mechanisms are based on making resource allocation as transparent as possible to client processes; they include: the ability of managers to redirect requests to other managers of the same type (i.e., peer managers), the ability of managers to accumulate information about the status of their peers, and the ability of users or applications to indicate preferred hosts. For example, a file manager answering a request to create a file may discover that it has little storage available. Instead of rejecting the request, it is redirected to another file manager with available storage. An example of resource management in the C<sup>2</sup> Internet application is the assignment of a newly identified target to an appropriate target manager.

In virtually all computer systems, access control mechanisms are necessary to prevent the unauthorized use of services and data, and to preserve system and component integrity; Cronus is no exception. In fact, robust access control techniques are particularly crucial in Cronus, since by design Cronus users have access to a potentially large number of hosts and facilities. All Cronus operations are subject to access control restrictions. Access control is provided in two phases: identification and authorization. When a Cronus process is started, it obtains a particular verified identity. Then, when operations are invoked on objects, the responding manager receives (from the operation switch) the identity of the invoking process. This identity is examined by the manager to see if it is valid for the object and operation requested. If so, the operation progresses; if not, it is rejected. For example, when a user session is started, its controlling process is associated with a user identity by the Cronus Authentication Manager using a user-supplied name and password. Future operations invoked by this process are authorized by the manager responding to the requested operation, using the user identity received for the process and an access control list associated with the referenced object. Obviously, the appropriate rights and who they are given to varies considerably for system and application objects. With C<sup>2</sup> Internet, we define appropriate rights for application objects as they are designed, and take advantage of existing mechanisms for the enforcement of access control.

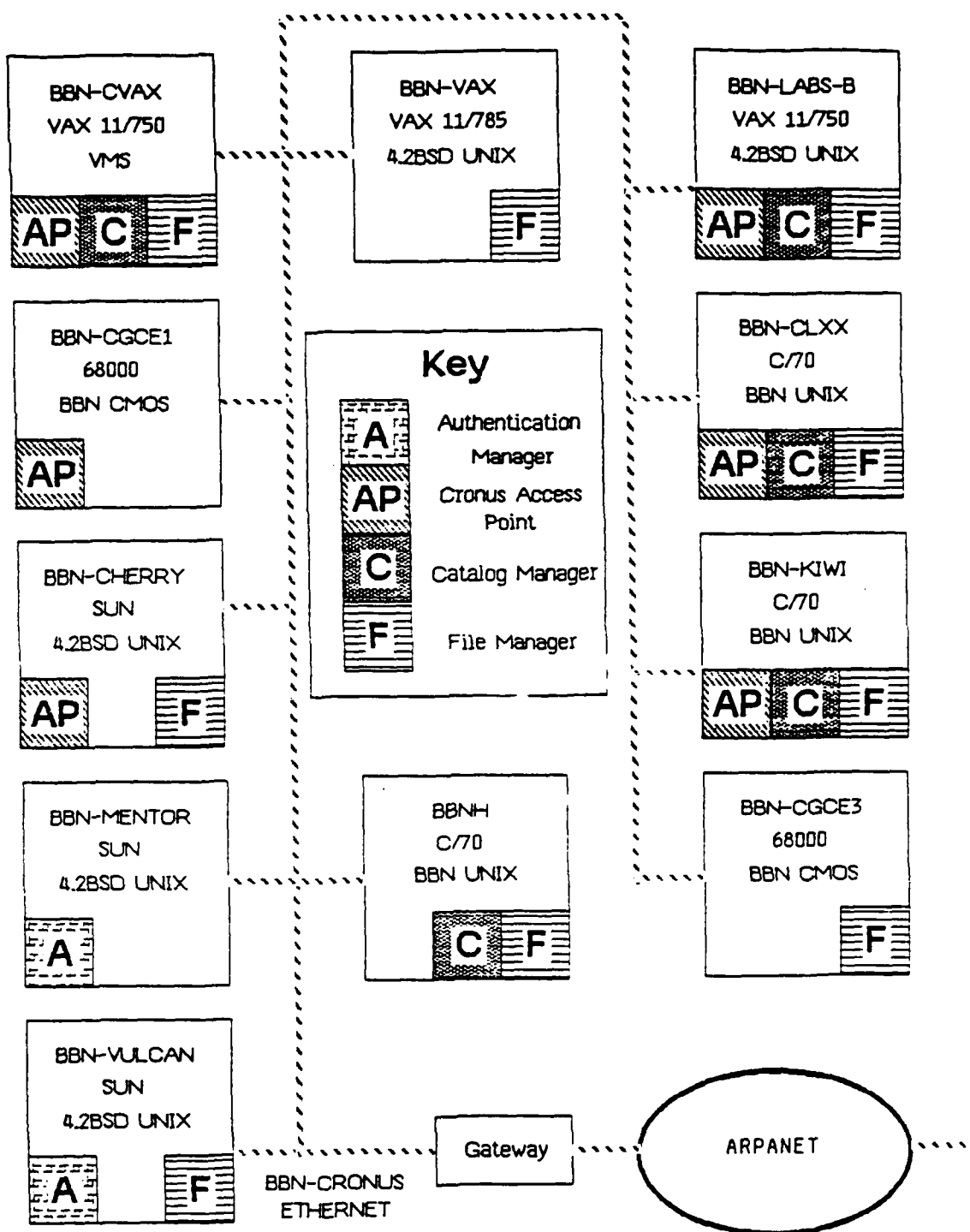
### 3.3. Cronus Core Functionality

In addition to the Cronus kernel, a number of Cronus managers provide the core functionality traditionally expected of an operating system. The resources provided by these services may be used both by other managers and by application programs. These managers are:

- **Primal Process Manager:** The Primal Process Manager manages Cronus processes. It provides a facility for remotely starting and stopping other managers, as well as a remote execution capability. The Primal Process Manager also maintains the access control information for each Cronus process.
- **File Managers:** Several types of storage objects are provided by a number of File Managers with varying characteristics (e.g., speed, survivability, etc.).
- **Catalog Manager:** The Catalog Manager provides the mechanism for the location-independent symbolic naming of any type of Cronus object. The Catalog Manager provides support for replicated directories to ensure that objects may be referenced symbolically despite system outages. In conjunction with file managers, the Catalog Manager provides the functionality of a traditional file system (i.e., access to symbolically named storage objects).
- **Authentication Manager:** The Authentication Manager implements the Principal and Group objects associated with the various identities and roles of the users of the system, and performs user validation functions. This manager, along with the abstract access control rights associated with each type, the *access control list* associated with each particular object, the collection of identities and affiliations associated with each process, and the automated checking of access rights during operation invocation provides a uniform Cronus access control system.
- **Constituent Interface Manager:** The COS Interface Manager allows local host files and directories to be manipulated as Cronus objects. This facility is particularly useful for integrating existing local resources into distributed applications and for providing a mechanism for accessing resources using both Cronus and constituent-based applications.

One Cronus cluster is presently in operation; a subset of the Cronus manager configuration in that cluster is shown in Figure 7.

For the C<sup>2</sup> Internet application, the process manager provides support for application processes and for remotely starting and stopping managers. Cronus files are used to archive data generated by the sensors. The catalog is used to support global symbolic names for experiment timers, targets, sensors, and other objects when the object must be identified by a user, as is the case when the experiment is being configured. The authentication manager is integral to the access control mechanism; this mechanism controls access to sensor data, cartographic data and the resource and logistics data. Access to the simulation control operations is also restricted using existing mechanisms. Finally, COS files are used for recording and processing large amounts of data when there already exists a program which runs on a constituent host for performing some application-related task.



BBN Cluster Configuration showing Cronus System Services  
Figure 7

### 3.4. Cronus Monitoring and Control

An overall view of system operation is provided using the Cronus Monitoring and Control System (MCS). The MCS passively monitors the status of various system components, and controls various parameters affecting the management of system resources. The MCS monitors both object managers and the Cronus kernel in order to provide a complete view of the operational system.

The data collected by the MCS is recorded, producing time-series data for later examination and evaluation. Both the instantaneous and aggregated data can be displayed to a system operator in graphical format. The MCS operator interface also supports invoking operations on both system and application components, and allows input parameters to be selected by *picking* values from a screen display.

The difference between application and system monitoring and control is largely in the training of the person operating the MCS and the ways in which data is presented for review, not a matter of how the data is collected or how control is accomplished. However, to understand the effects of Cronus behavior on the application, an operator must understand how the application uses Cronus resources. At the application level, we monitor the status of and control the managers and other components that constitute the application itself. At the lower level, we monitor and control the Cronus system components that support the application; this includes the Cronus interprocess communication mechanisms, the process manager, the catalog manager and other system components. This distinction is really only one of how familiar the components are to an operator of the system and of who can sensibly deal with situations requiring operator intervention. Components at the application level will be readily familiar to an application operator who understands the C<sup>2</sup> Internet Experiment architecture. Such a person, with experience, will also understand how those components interact.

### 3.5. Cronus Support for Design

In summary, Cronus supports the development of distributed systems and applications by providing:

- A *uniform, consistent system architecture* which addresses many distributed system issues, and the realization of this architecture with an implementation on a variety of computers.
- Support for commonly used *system functions*, such as naming, storage, and processing.
- Facilities for *system monitoring and control*.
- *Software development tools* to support the development of additional object managers and *user interfaces* to simplify the interaction with these managers.

#### 4. SYSTEM DESIGN DETAILS

The purpose of this section is to discuss the design of the  $C^2$  Internet application and, more generally, to describe the Cronus application design process. The application design precedes and hence provides direction for an implementation. Within the context of the present application, the design should insure that the planned experiment objectives are met, as well as insure that existing Cronus mechanisms are fully utilized and evaluated. Although the emphasis in this section is on the design of the planned experiment, the design process used models the design of other Cronus applications as well. One objective of building the  $C^2$  Internet distributed application is to further gain insight, from the application perspective, into the development of complex distributed applications.

In the present design, some of the application subsystems were selected for *stubbing*. By stubbing we mean a simplistic design and implementation of the actual subsystem desired. A subsystem may be stubbed for a variety of reasons:

- The software in question is not particularly relevant to Cronus evaluation or application demonstration. That is, the contribution to the experiment's goals does not warrant the software development cost of a full-fidelity subsystem.
- The relevant subsystem would most appropriately be constructed using anticipated Cronus enhancements or extensions not yet available, e.g., a database management capability.
- The subsystem serves as a placeholder (to support testing and integration of related components) for a component which may be independently developed at a later date or is not yet scheduled for full-scale implementation.

Application stubbing during the early stages of development is consistent with the Cronus concept of system evolution. The system is thought of as dynamic: changing, growing and improving as technology advances.

In the remainder of this section, we initially discuss overall design issues, and then discuss the design of specific components of the  $C^2$  Internet Experiment application. The material presented is indicative of the design process that must precede any serious implementation efforts. In an operational distributed application, however, the design process will often be more thorough than required here.

##### 4.1. Functional Decomposition

The preliminary decomposition of the proposed experimental system emphasized the functional aspects of the individual constituent elements. For example, individual components such as a remote sensing system, cartographic data management system, sensor data collection system, resource data management system, and mission planning support system were identified. Two distinct monitoring and control functions were also identified: one for monitoring and controlling the experiment in real time, and the other for monitoring and controlling the distributed computing environment of the application. This decomposition was shown in Figure 2. The functional boundaries selected for this experiment isolate constituent elements that can be independently developed and then integrated, achieving varying degrees of coupling.

The partitioning along functional boundaries is both a common and natural approach, and well suited to many of the benefits anticipated from a distributed system architecture. For example, the evolution of an existing system for the purpose of improving performance and capability might require the replacement or addition of system functions and/or hardware. Such functions could be developed independently and then integrated into the existing system.

The next level of design decomposition involved the distribution of the functional components to available computing resources. Two clusters of hosts will be used for the C<sup>2</sup> Internet Experiment. The functions described in Section 2 will be implemented and distributed over these clusters and hosts. Presently one cluster is operational at BBN. The second cluster is being established at RADC, Griffiss Air Force Base. The distribution of the application data sources and processing elements will be arranged to fully test the attributes afforded by a distributed system architecture within the constraints of the communication networks involved. The initial multicenter assignment is shown in Figure 8. This decomposition attempts to utilize the networks efficiently, and is based on the concept that the application processing elements within a cluster will be functionally related.

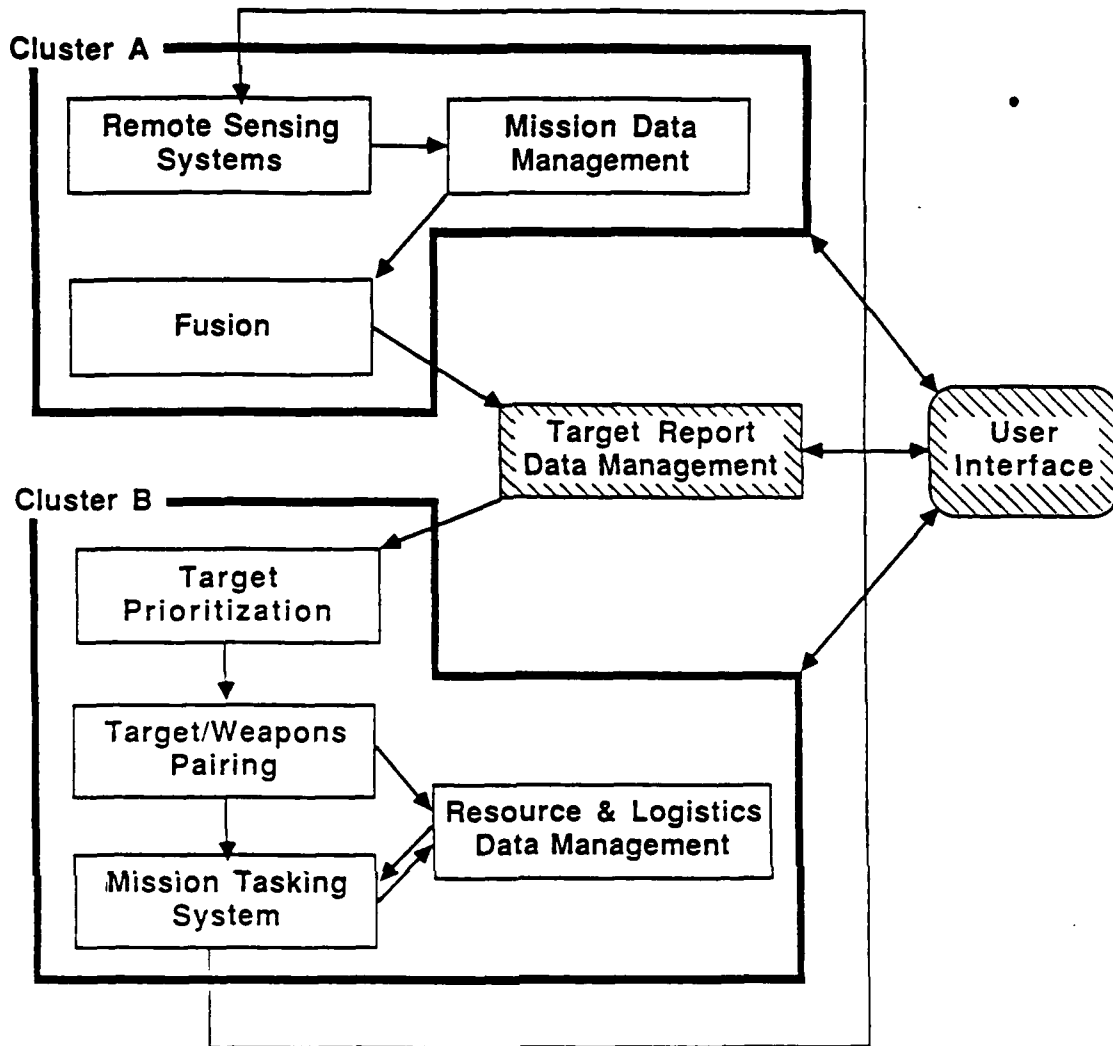
Next, host machines within a cluster were selected for implementation of the various functions. The emphasis at this time is on the allocation of the machines within the BBN cluster. The extension to the RADC cluster will take place when the appropriate resources become available. This will provide a valuable opportunity to demonstrate and evaluate the versatility of the Cronus concepts supporting system evolution and multicenter configuration. The allocation of functions to specific machines is not intended to be a hard and fast assignment. In fact, as the planned experiment evolves, changes to the allocation of hosts are inevitable. Two main reasons for this are the addition of new machines to the existing BBN cluster, and the inclusion of the RADC cluster and appropriate wide-area networks into the experiment. The inherent portability of most Cronus programs greatly increases the flexibility in choosing host assignments for various system elements.

#### 4.2. Design Details

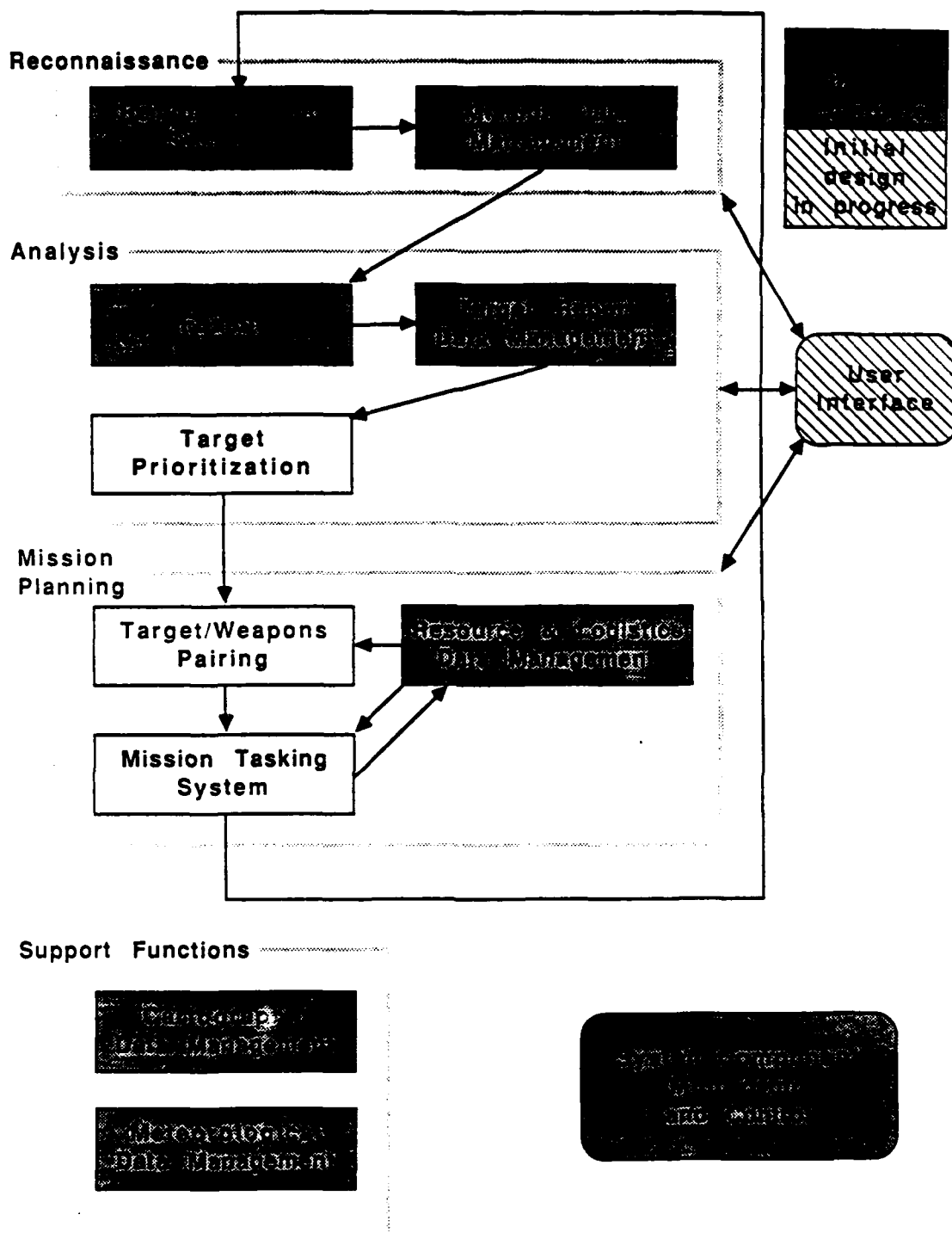
The remainder of this section contains a detailed description of the experiment components which have been designed to date. This corresponds to the portion of the system shown in Figure 9. Although this represents only a portion of the entire system, it is already sufficiently complex to warrant the need for access control, resource management, survivability, etc. It also affords the opportunity to test and evaluate the Cronus mechanisms available for distributed system development. The insight gained during this development period will be used extensively during the subsequent development of the remainder of the system.

For each functional unit in the application, the design will be discussed using the following organization:

- A description of the function and its role in the application.
- A discussion of design-related issues pertaining to application and experiment objectives.
- A description of the mapping of the function into the Cronus model; that is, a definition of the object type(s) and operations needed.



Multicenter Assignment of C<sup>2</sup> Internet Functional Elements  
Figure 8



Overview of Experimental C<sup>2</sup> Internet System Currently Designed  
Figure 9

- A discussion of the relationship between the design objectives and the object type(s) and operations defined.
- A discussion of any additional design-related and experiment-related issues.

The various functional elements described fall into three basic classes: prototype, simulation, and experiment control. A prototype subsystem is a functional element that might be found in an actual command and control system in a form very similar to that described here (although probably in a more complete design and implementation). An example of such a subsystem is the Meteorological Data Management System. A simulation subsystem is used to emulate the characteristics of some functional element of an actual command and control system. An example of a simulation subsystem is the Sensor Manager which emulates the operation of an individual Remote Sensing System. Finally, an experiment control subsystem is a functional element that exists to support the experimental environment. The Target Simulator is an example of such a subsystem.

#### 4.2.1. Timer Manager

Early in the design phase, it was determined that an experiment clock was needed in order to control and synchronize the progression of events during the experiment. To achieve this control and synchronization, the experiment time must be available to the various managers and clients distributed throughout the multicluster configuration.

A simulation timer provides a common time to all managers and clients so that a coherent view of the  $C^2$  Internet simulation can be constructed. A manager or application program determines the current simulation time by sampling a simulation clock, which, from the client's perspective, appears to be a single object provided by the timer service. The manager or application may then collect information from other managers, and thereby construct a temporally consistent view of part of the simulated environment for analysis or for presentation to users or client programs. For example, a sensor performs surveillance over a time interval using a mission schedule. By using this mission schedule in conjunction with the experiment time, target detections in accordance with the planned progression of the experiment are generated.

The simulation timer regulates the progress of the  $C^2$  application. Thus by changing the rate at which a simulation timer runs, we can manually control the progress of the overall application simulation. The experiment clock can also be stopped at any time. Additionally, in order to simplify program development, we provide multiple, independent simulation timers; this allows independent developers to develop, test, and debug various subsystems independently.

After identifying the need for an experiment clock and determining its desired features, the Cronus model and mechanisms are applied to its design. The timer object type was devised to support these requirements. The following information characterizes each instance of a timer object.

- *Current Time* specifies the current simulated time.

- *Rate* specifies how many seconds of simulated time elapse for each second of real time.
- *Running* specifies whether the timer is running or not. Conceptually, a timer that is not running has an effective rate of zero; if it is restarted at some later time, the rate reverts to the value maintained by the rate parameter.

The following set of operations control timer objects.

- *Create* a new timer.
- *Get* the current experiment time of the timer.
- *Set* the current experiment time of the timer.
- *Set* the rate at which the timer runs.
- *Start* the timer.
- *Stop* the timer.

For convenience, a few operations which allow a user to control the timer directly were added. These additions allow the experimenter or software developer to reset the timer to an initial value, set a stop time so the timer will run to a specified time and then stop automatically, or step the time by a specified interval. These additional operations are:

- *Set reset value* specifies the desired setting of the timer after a reset operation.
- *Reset [value]* resets the experiment time. The default for the optional value is set with the *set reset value* operation.
- *Set stop time* causes the timer to stop automatically when the specified time is reached. The stop time can be specified as an optional parameter on the start operation as well.
- *Set step interval* allows time step to be set or changed.
- *Step [interval]* advances the timer by one step. The default for the optional interval is set with the *set step interval* operation.

Since the timer is required for the continued operation of the simulation, it must survive host failures. Further, since it is widely used, the cost of sampling the timer in our design must not dominate the overall cost of running the simulation. Both of these requirements lead to the decision to replicate the timer manager across several hosts. Properly done, this ensures continued availability and reduces the overhead of sampling the time since local timer managers are used and the load for sampling is distributed across several hosts.

The need for replication occurs in two forms. First, the state information associated with each timer must be replicated so that changes performed by one manager are propagated to its peers. In addition, a synchronized source of the real time is needed to ensure that matching time values are returned by independent managers.

While the details of how this is carried out are left to the implementation, it is important to note that we allow the times supplied by various timers to vary by a few seconds since each manager constructs its view of the simulated world by sampling the current real time from the local system clock and then deriving the experiment time from that time.

The accuracy of experiment synchronization depends, in part, on the frequency with which timer managers compensate for discrepancies in local host times. In our design this period is adjustable by an operator. However, care is needed to insure that the frequent resynchronization of the timers does not unnecessarily increase the cost of maintaining the experiment time.

We believe that a synchronization error of a few seconds is tolerable for this application, and that this level of synchronization can be provided at a minimum cost in overall experiment resources.

It is important to note that the timer manager is both a tool for the  $C^2$  application and a more general support capability. All distributed simulations require coordinated behavior which can be achieved through simulation timers. Maintaining a global time base is also a general purpose synchronization tool with a variety of applications.

#### 4.2.2. Target Simulation

A number of components were constructed to simulate the action of real  $C^2$  functions. Each Remote Sensing System (see Figure 2) is composed of two components, which together emulate the operation of a sensor system. These components interact with the Timer Manager to synchronize with the rest of the experiment.

In order to drive the command and control application, there is a need for a function that simulates the movement of targets. These simulated targets move in accordance with a prescribed scenario built up of concatenated track segments. Each target is represented as a Cronus object of type *target* and is characterized by the following.

- A *name*.
- A *target type* i.e., jeep, tank, etc.
- A series of *track segments* which represent the movement of the target over time.

We require the following operations to manipulate target objects:

- The *create* operation produces a target with the specified name and type; an initial position for the target and the time at which it can first be observed in the scenario are also specified. This operation assigns a unique identifier (UID) for subsequently referencing this target instance. The target instance is also cataloged using the target name specified, which provides a user-oriented mechanism for naming specific targets. This global symbolic name for the target is maintained by the Cronus Catalog Manager.

- The *moveto* operation adds a segment to the target trajectory. The segment begins at the end position and time of the previous segment and ends at the specified new location and time.
- The *targets in region* operation returns, to the requesting client, the target segments that fall within the specified region and time-interval: a region is denoted by two pairs of latitude and longitude coordinates which define the Southwest and Northeast corners of a rectangular area. The time interval is denoted by the beginning time and ending time specified.

The following example demonstrates how a single target track would be set up.

```
CREATE jeep j10 N50deg00'00" E10deg00'00" 19:00:00
MOVETO j10 N50deg01'00" E10deg02'00" 19:10:00
MOVETO j10 N50deg03'00" E10deg02'00" 19:21:00
MOVETO j10 N50deg07'00" E10deg01'00" 19:57:00
```

By creating a number of simulated targets, a complete scenario such as the one described in Section 2 can be constructed. Composite targets, such as a battalion level force, may be simulated using a collection of individual targets.

#### 4.2.3. Sensor Manager

One of the primary functions selected for the planned  $C^2$  experiment is ground target situation assessment. Therefore, part of the design focuses on the simulation of remote sensing systems that sense ground target activity from an airborne platform. The quality of the simulated sensor data, which is intended for use in the situation analysis process, should depend upon such factors as sensor technology, target type, and environmental characteristics such as terrain and weather. The goal of the present design is to capture the salient aspects of sensor systems for supporting the rest of the  $C^2$  Internet Experiment with a minimal amount of software development.

Functionally, the sensor simulation can be viewed as follows. The control of a specific sensor is maintained through a mission schedule: a planned list of surveillance regions and associated times for a specific sensor. The mission schedule can be generated and modified during the course of an experiment by authorized personnel from anywhere in the Cronus system, subject to appropriate Cronus access controls. This mission schedule is used in conjunction with the experiment time to generate simulated sensor output data. As the experiment time progresses, the mission schedule is examined and the target data appropriate for the surveillance region and time is acquired by the sensor from the target simulator. The target data is then transformed into a simulated view of ground activity in accordance with the sensor characteristics and other attributes mentioned above. The raw simulated sensor data is recorded locally for archival purposes, while a concise summary of detections is stored remotely in a multihost repository.

The Cronus system model forms the basis for the design. We define a *sensor* object type, which (for the experiment) is completely characterized by the following.

- A sensor *type* (presently, radar or infrared).
- A sensor *name* (e.g., Radar-1),
- The sensor's *resolution* (e.g., 1 degree of solid angle).
- A (mission) *schedule* consisting of a sequence of regions and time durations for surveillance.

The following operations were identified as pertinent to the sensor simulation:

- *Create* an instance of the sensor object on the host local to the manager. The sensor created will be characterized by name, type, and resolution.
- *MoveTo* adds a mission to a specific sensor's schedule.
- *ShowSchedule* retrieves the schedule for a particular sensor.
- *Edit* allows alteration of the description of a particular sensor.
- *Operatesensor*, performs the sequence of procedures necessary to generate real-time sensor data for experiment purposes.
- *GetDetections* returns the sensor's target detections for the current mission up to the present (simulation) time.
- *SetDetectionThreshold* adjusts the threshold above which a sensor reports a target that is detected. (Below the threshold, the sensor interprets the data as noise only.)

A multisensor scenario is planned for the experiment. To realize this configuration, independent sensor simulations will run concurrently on different hosts distributed throughout the multicluster configuration. Each will perform in accordance with a prescribed sensor characterization and mission schedule. Time synchronization will be maintained by the experiment clock facility and consistent target conditions will be coordinated through the simulated target manager.

#### 4.2.4. Mission Data Manager

Sensor data, produced by each sensor as it carries out its mission, must be available for use in analysis and review. The majority of these analysis tasks emphasize numeric and symbolic processing techniques and work best when applied to the information collected from several sensor missions. By providing a logically centralized service to provide information from all sensor missions, and providing a simple query facility for retrieving this information, the Mission Data Managers simplify the task of developing the automatic processing components. The Mission Data Managers both provide a single point of contact for requesting sensor data, and insulate clients of the data from the process of carrying out data queries: identifying the sensors with relevant data, collecting the data, and normalizing its format. Storage of the detection data by the Mission Data Managers also improves the availability of detection data, since the Mission Data Managers are much less vulnerable to damage or loss than the sensors themselves. The raw sensor data, which is much more voluminous and which is less often used in analysis, is available only from the sensors.

For each mission, the following information is stored by the Mission Data Manager:

- *Sensor ID* identifies the sensor which is reporting the data.
- *Threshold* value reports the *signal-to-noise* ratio threshold which is used to distinguish events from noise in the data set.
- *Mission Number* identifies which of the sensor's scheduled missions acquired the data.
- *Sensor Variable* describes various characteristics of the sensor.
- *Mission ID*, generated by the Mission Data Manager, uniquely identifies each data set.

The data records for each mission are stored as a Cronus object. Thus, the request to start data recording is just a request to create a Mission Data Object. A round-robin resource management algorithm is used to select the location for the new object. (In round-robin, managers are assigned a circular, numerical ordering and missions are assigned to each manager in turn.) Discarding old data is done by removing the appropriate mission data objects. Adding new data records to a recorded mission requires invoking an operation on the mission data object, created for use by the sensor, and including the time the sensor data sample was taken and a list of detections. Each detection gives the following information:

- *Location* gives the latitude, longitude and altitude where the detection is thought to be positioned.
- *SNR* gives the signal to noise ratio of the detection, which measures how strongly the detection is being received.
- *Target ID* gives additional information about the detected object.

Detections lists are requested by clients through queries submitted to any Mission Data Manager. Each request identifies characteristics of the desired detections. Each request must specify the geographic region which should be examined for detections, and may optionally specify the time range and reporting sensor ID. The manager performing the request submits similar queries to its peer managers and then combines all the results; thus data from all relevant mission objects will be returned to the client.

#### 4.2.5. Fusion (Detection Correlation) Processing

In its initial version, the detection report fusion processing will be performed by a simple client program that uses cues in the *target id* field of each detection to eliminate false alarms, correlate detections provided by different sensors, and identify target trajectories. This program will request detection lists from the Mission Data Managers, perform the correlation processing, and submit the resulting target reports to the Target Report Manager for later review and action by system users.

#### 4.2.6. Target Report Manager

Suspected targets, detected by the sensors and verified by the fusion processing, are recorded by the Target Report Manager. Each target report, stored as a Cronus object, summarizes the information known about a detected target. This information includes:

- *Name*, selected by a user, mneumonically identifies the target.
- *Type* identifies the kind of target, such as tank or aircraft.
- *Destination* is the known or projected destination of the target.
- *Priority* measures the relative importance of the target.
- *Weapon* identifies which weapon, if any, has been assigned to deal with the target.
- *Start Time* specifies when the target was first detected.
- *End Time* specifies when the target was last detected.

The information in these reports is initially provided by automatic processing components, and may be incomplete. Later review by human analysts may lead to changes in the information, or the addition of fields that could not be assigned automatically. Requests on objects of this type include:

- *Store Target Report* creates a new target report. The initial description of the target is included in the request.
- *Get Target Descriptions* returns a list target reports for all detected targets in a particular region. A time range may also be specified.
- *Get Target Report* returns the description of particular target.

#### 4.2.7. Meteorological Data Manager

Convenient access to meteorological data enables mission planners to choose appropriate sensor systems and weapon resources to match chosen targets in varying weather conditions. It also provides information to personnel performing target data correlation on the relative accuracy of various sensor types (since sensor performance is affected by varying weather conditions).

The Meteorological Data Management System provided by one or more meteorological data managers allows a meteorologist to enter or modify meteorological data for later retrieval by other personnel or client programs. In designing the meteorological data manager, two basic object types were identified, *weather reports* and *weather forecasts*. Weather reports contain observations of existing weather; weather forecasts contain projections of weather for the future.

An instance of a weather report object is characterized by the following attributes.

- A *location* specified using latitude and longitude coordinates.

- The *time* at which the report was made.
- *Report data* summarizing the observed weather conditions.

We require the following set of operations to manipulate weather reports:

- *Create* produces a new report.
- *ShowWeatherNearLocation* retrieves the most up-to-date report for the location nearest to the location specified.

An instance of a weather forecast object is characterized by the following attributes.

- A *location* specified using latitude and longitude coordinates.
- The *time* at which the forecast was made.
- The *validity period* for the forecast specified using a *beginning time* and an *ending time*.
- *Forecast data* summarizing the predicted weather conditions.

We require the following set of operations to manipulate weather forecasts:

- *Create* produces a new forecast.
- *ShowForecastNearLocation* retrieves the most up-to-date forecast for the location nearest to the location specified, and for the appropriate validity period.
- *DeleteOutdatedForecasts* disposes of old, unneeded information.

#### 4.2.8. Cartographic Data Manager

Handling cartographic data is an important part of almost any  $C^2$  application. For example, cartographics can serve as a user interface aid in the planning of a sensor mission schedule, in a ground force situation analysis, or in a target track prediction function. Thus it is desirable to provide a uniform but general-purpose mechanism for clients to obtain cartographic data.

Our initial design for this part of the system follows. Cartographic data is viewed as a collection of features. Each feature is a descriptive element of the area of interest; a forest, a road, and a bridge are examples of features. A collection of the cartographic features in a given geographic region is referred to as a *feature map*, which is the basic object type managed by the cartographic data manager.

An instance of a feature map object is characterized by the following attributes.

- Two *locations* (specified using latitude and longitude coordinates) denoting the southwest and northeast corners of the region represented by the feature map.

- A list of *feature identifiers* identifying features that are contained in the feature map (e.g., forests, lakes, roads, buildings).
- An array of *locations* (latitude and longitude) for each feature describing the extent of that feature.

We require the following set of operations to manipulate feature maps.

- *Create* a new feature map.
- *ShowFeaturesInArea* which retrieves the features in a specified region. This may entail retrieving all of the features, or specific kinds of features. In the latter case, the manager filters out those features not of interest to the client before responding to the request.

#### 4.2.9. Resources and Logistics Data Management System

In the C<sup>2</sup> Internet Experiment, the availability of resource and logistics data will assist in the target/weapons pairing and mission planning functions. The Resources and Logistics Data Management System maintains two kinds of information: capability and availability. Capability information contains, for a given resource, the characteristics which appropriately summarize the resource's performance. For example, the capabilities of an airfield would include airfield location, runway lengths and directions, etc. An aircraft could be described using such characteristics as maximum speed, maximum range, takeoff length, fuel capacity, etc. Availability information details the readiness of specific resources. For instance, the location and quantity of a specific type of aircraft.

Convenient access to capability and availability data enables mission planners to make effective use of those resources at their disposal. The Resources and Logistics Data Management System (provided by one or more managers) allows capability and availability data to be entered (or modified) for later retrieval by authorized personnel or client programs.

One possible design for the Resources and Logistics Data Management System would be as a collection of managers for a hierarchy of object types. Each type would represent either capability or availability information for the appropriate resources. A partial hierarchical type structure for capability data might look like the following:

Resource  
  Equipment  
    Armament  
    Remote Sensing System  
    Vehicle  
      Aircraft  
        Fighter  
        Bomber  
        Reconnaissance  
      Truck  
        Mobile Command Post  
        Fuel  
  Facility  
    Airfield

At present, a limited version of the resources and logistics data manager maintains capability data for the *vehicle* object type. (In its present state this capability data is oriented toward aircraft characteristics.) This initial design is intended to demonstrate a limited database-like functionality within the application.

An instance of a vehicle object is characterized by the following:

- The *name* of the vehicle.
- A list of *capabilities* (e.g., maximum speed) for the vehicle.

We require the following set of operations to manipulate vehicles:

- *Create* a new vehicle.
- *ListVehicles* to list all of the known vehicles.
- *ListVehiclesWithCapability* to list those vehicles having particular characteristics (e.g., a certain minimum speed).
- *ShowCapabilityForVehicle* to show the characteristics of a particular vehicle.

#### 4.3. Design Evolution

In this section, we have described those C<sup>2</sup> Internet Experiment components designed to date. The application was decomposed along functional boundaries. Each of the functional elements was then analyzed in terms of Cronus support mechanisms. It is expected that these components will evolve as additional development of the application and Cronus takes place. We also continue to design and integrate additional components into the experiment architecture as outlined in Section 2.

In the following section, we will discuss the implementation details for those components whose design has been outlined. The implementation will, we expect, feed back into the design process, both for the C<sup>2</sup> Internet application and Cronus itself, resulting in an evolution of both of these elements. Such insight and feedback is especially relevant at this time, due to the limited application experience in the development of complex distributed systems. In addition, initial hands-on experience with a small set of integrated components will improve our understanding of the operational characteristics of Cronus.

## 5. SYSTEM IMPLEMENTATION

In previous sections, we have described the design phases of developing the  $C^2$  Internet distributed application. There, we identified the functions the system had to perform, refined the definition of the functions by designing objects and operations on the objects, and identified requirements of that application, such as the degree of survivability for particular functions, that must be satisfied through the interaction of the objects. To reduce the complexity of those pre-implementation phases, in general, we try to avoid in-depth consideration of algorithms, performance, and existing support code. However, these considerations become a primary concern for the implementation phase.

During the implementation phase we:

1. *Formalize specifications* for objects and operations developed during the earlier phases;
2. *Choose specific representations* for the objects and for the parameters that are relevant for each operation; and
3. *Implement the code* using the Cronus system and its software development tools as a support base.

Our primary goals in this phase are to produce an implementation which is sufficiently thorough to form the basis for an evaluation of the quality and suitability of the underlying Cronus support for distributed application development, and to demonstrate Cronus' operational capabilities. This effort is also being used to direct refinements of the Cronus distributed systems development approach and modifications to the facilities Cronus provides.

### 5.1. Implementation Strategy

Although the scale of the  $C^2$  Internet Experiment is more limited than that of an actual  $C^2$  system, it reflects most of the complexity of managing development in a  $C^2$  environment. The experiment involves several computers, employs components whose interaction reflects the interactions of the components of a real application, and is being implemented by several developers. It is particularly important that developers be insulated from hardware and software changes made by other developers as existing hardware and software components are added or evolve as improved versions become available. Throughout the experiment, implementation has deliberately been structured incrementally. An incremental approach more closely resembles a natural product life cycle and offers an opportunity to evaluate the effectiveness of the Cronus object model and support for this approach. In addition, from the application perspective, the implementation of initial components provides early experience with such issues as sizing, performance estimation and user community exposure.

We have completed initial versions of the components that form the a significant subset of the  $C^2$  application. This set of components was selected for initial implementation because they are sufficient to provide a rich, self-contained environment capable of supporting non-trivial demonstrations. Additional components can be added to the system, and the existing implementation provides a framework for providing simulated input data, a user interface for entering commands and examining results, and monitoring and control facilities for monitoring component status and controlling the progress of the

experiment. The components which have been implemented include the following: *Timer Manager* for control and synchronization of an experiment simulation clock; *Target Manager* for simulation of real-world targets; *Sensor Manager* for simulation of remotely sensed surveillance data for radar and infrared systems; *Meteorological Data Manager* for generation of simulated weather conditions and reports; *Cartographic Data Manager* for manipulation of available "standardized" cartographic data; and *Resource and Logistics Manager* providing remote access to information about the availability and capabilities of various C<sup>2</sup> application resources. This subset is depicted in Figure 10.

## 5.2. Component Implementations

Subsystem implementation is the subject of the remainder of this section; descriptions of each subsystem implementation follow. We also have included numerous examples which help to illustrate the present state and use of the experiment implementation.

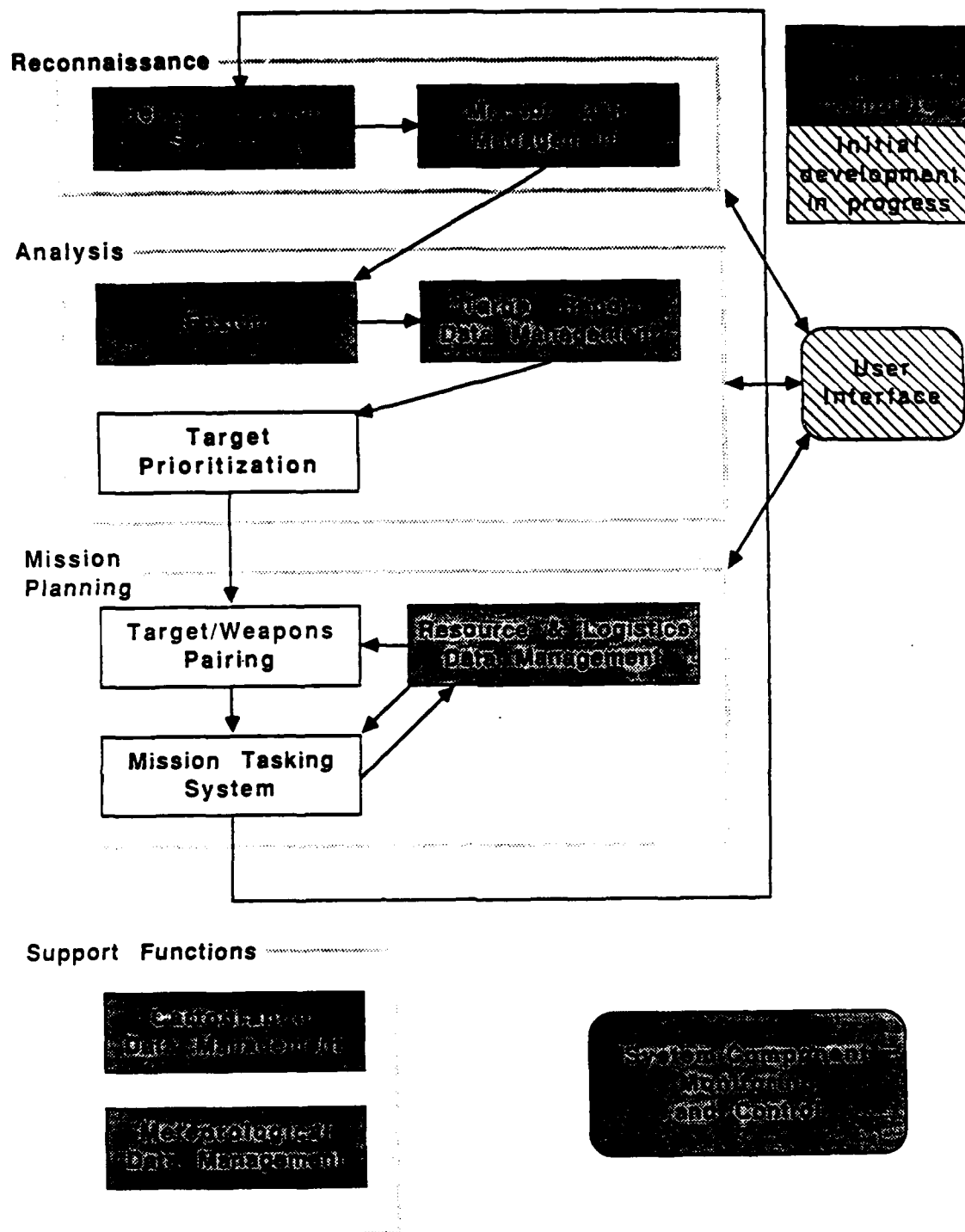
### 5.2.1. Timer Manager

The timer manager provides a survivable source of system-wide experiment timers. The current implementation derives the value of an experiment timer by transforming the real time of day in conjunction with rate and reference values recorded in the attributes of the particular timer object. The time of day clocks are periodically synchronized to correct for the variation between processor clocks.

Our implementation consists of two, loosely coupled parts: an upper layer which supports the operations on timer objects, and an underlying layer which provides a system-wide, synchronized time of day used by the timer objects, and which synchronizes the attributes of the replicated timer objects among their instances.

We implemented an initial version that supported only the upper layer, on a single host, using the local processor clock to provide the time of day; this offered an immediate source of experiment time to other C<sup>2</sup> application managers. We then added the lower layer to provide survivability and more immediate accessibility. The conversion did not require any major changes to the existing manager code since the replication strategies are largely separate from the functions and algorithms used to support the timer operations. Adding replication was not visible to clients, since the replicated manager provides the same client interface and the underlying Cronus support makes the cardinality of the manager logically transparent to the clients. The remainder of this section describes how each of these layers was implemented.

When the timer is running, the current experiment time is derived from the last time setting made by a user, the time of day when that setting was made, and the rate at which the timer is running. Thus, if ten minutes ago the timer was set to time  $T$ , and the rate is 2, the time is now  $T+20$  minutes. Similarly, if the rate had been  $1/2$ , the timer value now would be  $T+5$  minutes. This is expressed formally as:



Overview of Experimental C<sup>2</sup> Internet System Currently Developed  
Figure 10

```

if TimerRunning
then
    ElapsedTime = RealTime - ReferenceRealTime
    TimeValue = ElapsedTime * Rate + ReferenceExpTime
else
    TimeValue = ReferenceExpTime
end

```

Setting the value of the timer then consists of setting *ReferenceExpTime* to the new timer value and setting *ReferenceRealTime* to the time of day at the moment the reference experiment time is set. Thereafter, calculating *TimeValue* produces the experiment time.

Special treatment is required whenever starting, stopping or changing the rate of a timer: the timer's value must be preserved. This is done by calculating the experiment time before changing the parameters, then changing the parameters, and finally updating the reference experiment time to the time calculated before the parameters were changed.

A few additional object attributes support the remaining operations. A *stop time* is used to record when the timer should stop running; if we sample the timer and discover that we have passed the stop time, we set the reference time to the stop time and set *TimerRunning* to false. We store a *reset value* which we use to set the timer whenever the reset command is issued. We store a *step increment* to handle the step mode of running the timer; whenever the step command is invoked, we set the stop time to *TimeValue + StepIncrement* and start the timer.

Although we were not bound to use Cronus canonical types in our implementation, we chose to use them for portability and because it would simplify our transition to replicated timer objects later. We simply include a description of the object attributes, similar to that appearing in Table 1<sup>2</sup>, along with the operation specifications we provide to the Cronus application development tools. The tools, in turn, generate code to handle storage and retrieval of the timer objects, and code to dispatch operations to operation subroutines we provide. We complete the code for the manager by writing these subroutines in C. Since the Cronus application development tools generate appropriate code for each of our target machines and the C code we write can be transported among our target machines unchanged, the timer manager can be easily transported from one machine to another. In addition, specifying that the timer objects should be replicated in the manager specification activates Cronus library software that handles the synchronization of replicated objects.

The survivable timer required two forms of cooperation between instances of the manager, one to replicate the attributes of each timer object and another to correct for drift between the time of day clocks on various hosts. We use the Cronus application development tool's support for automatically handling object replication for timer attribute replication, as we mentioned earlier. For synchronization,

<sup>2</sup>The primitive types that are provided by Cronus software support and the routines that implement them are described in the Cronus User's Manual [BBN-6180] and the Cronus Programmer's Manual [BBN-6181]. Here, we have used three different canonical types. For time values, we chose the Cronus primitive canonical type *EDATE*. The *EDATE* is used to represent time values; for each host, Cronus library routines are provided for calculations involving these values and for reading and printing these values. The *U16I* provides a 16 bit unsigned integer, similar to the *unsigned short* of C. And the *EBOOL* represents a boolean, with values of either TRUE or FALSE.

Attribute Name	Cronus Type	Description
ReferenceTime	EDATE	Last setting of timer
ReferenceRealTime	EDATE	Time when timer was set
Running	EBOOL	True when running, False when stopped
RateMultiplier	U16I	Numerator for rate of timer
RateDivisor	U16I	Demoninator for rate of timer
AwaitingStop	EBOOL	True if timer should be stopped at StopTime
StopTime	EDATE	Time when timer should stop
ResetValue	EDATE	Setting of timer used by Reset command
StepIncrement	EDATE	Interval timer will run for when step command given

Timer Object Attributes  
Table 1

we implemented a strategy that elects one of the managers, thereafter called the *master clock* as the primary source of time of day. If the master clock fails, another manager will be elected.

Timer object replication is handled in the following way. To modify the value of an object attribute, such as the *rate*, the manager retrieves the attributes for that object, changes the value, and then records the results. When the changes are recorded for a replicated object, the new attributes are instantly broadcast to other instances of managers for the same type. Managers which are responsible for copies of the modified object will then record the changes. Reintegration procedures for managers which have been temporarily inactive or unavailable, are also provided as part of the automated replication support.

The implementation for time of day synchronization relies upon selecting a *master clock*, which is then responsible for periodically reporting the time of day to the other *slave clocks*. The slave clocks, then calculate an adjustment to their local processor time of day clocks, which will be used to correct the local time of day value whenever it is used. Synchronizing the clocks every few minutes requires very little processing and corrects for short term drift between the time of day clocks on the various machines.

Election of a new master clock occurs when a timer manager is first started, when the master clock fails, or when an operator manually selects a new master clock. When a timer manager is first started, it asks for the time of day from its peers; if none answer, the manager assumes the role of master clock. Thereafter, whenever another manager is started, it receives the time of day from the master clock, becomes a slave clock, and expects to receive periodic synchronization messages from the master clock. A watchdog timer is used by the slave clocks to detect when two consecutive synchronization messages have been missed, which is assumed to mean the master clock has failed.

A slave clock becomes a candidate for election when either its watchdog timer expires, indicating that the master clock has failed, or when an operator selects the slave clock as the new master clock. The slave announces its candidacy by broadcasting a bid to the other managers. All candidates exchange bids. If the master clock is actually available, it will exchange bids with the candidates too. The winner is chosen by comparing the bids in a pairwise fashion.

- If one member of the pair was chosen by the operator, it wins. If both members were chosen by operators, the remaining attributes determine the winner.
- If one of the pair is the original master clock, it wins.
- If one has a higher, operator assigned quality factor, it wins; the operator assigns low quality values to managers on hosts with poor time of day clocks, and high values to managers on hosts with very accurate time of day sources, such as WWV clocks.
- If one of the pair has received a synchronization message more recently from the master clock, it wins.
- Otherwise, to resolve ties, the one with a higher host address wins.

If no counterbids are received, or if the manager wins against all counterbids, it will become the master clock. Otherwise, the winner of the bidding becomes the master clock. In either case, the new master clock begins to periodically broadcast synchronization messages. The remaining slave clocks then resume awaiting synchronization requests from the new master.

This strategy for assigning the function of master clock is an example of global resource management among the timer managers. Over time, we anticipate that this approach will be generalized for use with other resource management strategies.

### 5.2.2. Target Simulation

The major role of the Target Simulation Manager is to supply the target scenario selected. This scenario is used primarily by the various sensor system simulations, for experiment execution. The motion of a target trajectory is described by a series of a straight line segments. Each segment approximates the path of a target over a distance sufficiently short that significant turning maneuvers do not occur. Each target object also has an associated type and name. The target object attributes are shown in Tables 2 and 3. These objects can be used to represent an instance of a single element, such as a jeep, or a collection of elements of one specific target type associated with a convoy unit (e.g., a collection of jeeps travelling in proximity with one another). A composite target, such as an airfield or a battalion, consists of an appropriate collection of the target objects of the appropriate types. Associated with each target is a UID which uniquely identifies the object and is used by the underlying Cronus support to locate the object, mediate access control, store symbolic references in the catalog, and identify replicated copies.

The two operations pertinent to generation and modification of the target scenario are *Create* and *MoveTo*. *Create* produces a new instance of a target with specified target type, name, initial location and time. The *MoveTo* operation is used to add track segments to an existing target object. Another important operation performed by the Target Simulation Manager is *TargetsInRegion*. A client, such as

Object Attributes	Cronus Type	Description
Name	ASC	target name
TargType	TARGETTYPE	jeep, tank, bridge, power plant, etc.
Segments	array of SEGMENT	target track segments

Target Object Attributes  
Table 2

Attributes	Cronus Type	Description
StartTime	EDATE	beginning time
StartLoc	LOC3D	beginning location
EndTime	EDATE	ending time
EndLoc	LOC3D	ending location

SEGMENT Attributes  
Table 3

the sensor simulator, uses this operation to obtain a list of target track segments representing target activity in a specified geographic region over a specified period of time. This is a good example of the Cronus concept of moving a high-level processing request to the data, rather than moving the raw data to the remote processing element.

Presently, a target scenario is generated by invoking a sequence of *Create* and *MoveTo* commands. This can be done with the aid of the user interface utility described earlier. Alternatively, the commands for certain target scenarios can be stored in a command file, which, when executed, generates or modifies the target scenario by creating the appropriate target objects and initializing their associated target track segments. In the future, the process of scenario generation will be further aided through the use of cartographics in conjunction with the experiment clock and a display cursor.

### 5.2.3. Sensor Manager

Simulated sensor data is the major source of data for ground target situation assessment. Presently, sensor data is generated through the simulation of several remote sensing systems employing varying sensing technologies. The generation of simulated sensor data requires the interaction of various components since the data giving the experiment time, the weather conditions, and the actual target state must be collected from different managers. The integrated operation of these managers with multiple instances of the sensor managers, relies totally upon the Cronus system for communication and coordination. Although this represents only a portion of the C<sup>2</sup> application planned for implementation,

it has produced numerous preliminary conclusions. For example, it became clear that subsystems could be designed, implemented and integrated quickly; however, certain software development tools were changed in response to difficulties discovered in their early use.

During this phase of implementation and evaluation, we believe that a high-fidelity sensor simulation is not necessary for application demonstration or Cronus system evaluation, and software development effort is more valuable elsewhere. However, this subsystem is a candidate for integration of independently developed, high-fidelity sensor simulations at a later time.

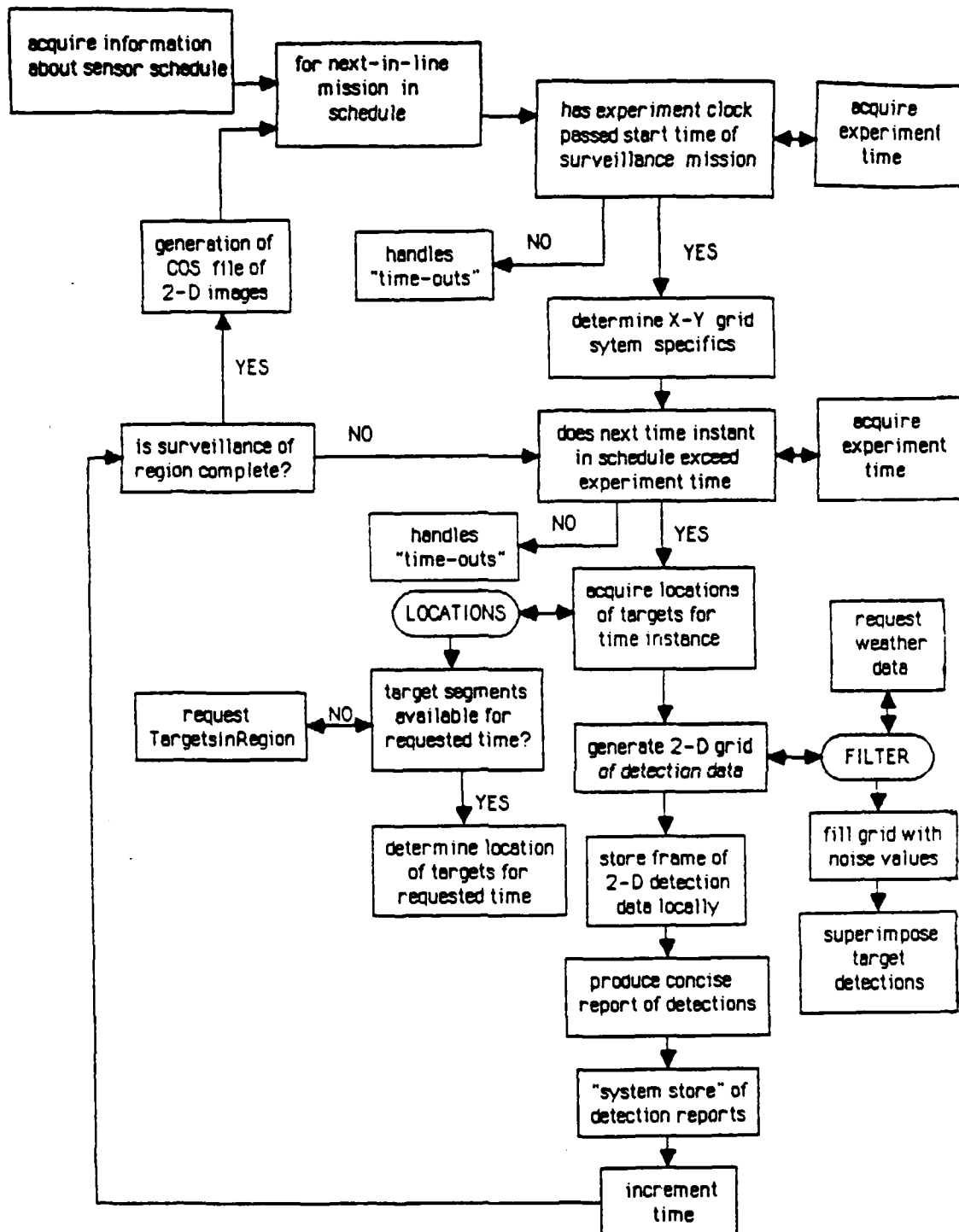
Although we have elected to limit the fidelity of the sensor emulation, we have made some effort to imitate the effects of environmental conditions on sensor performance. To accomplish this, the sensor manager uses a filtering algorithm to distort the view of the targets supplied by the target manager, as a function of weather conditions and the sensor type. Presently, only the effects of weather conditions are considered, and only for radar and infrared sensing systems. However, our design is well suited to adding sensitivity to other environmental conditions such as terrain, and to simulating other sensing technologies as well.

The functioning of an instance of a sensor can be described as follows. A flow chart of this process is shown in Figure 11. When the *OperateSensor* operation is invoked, generally by an application operator, to activate the sensor, the manager examines the specified sensor object's attributes to determine the sensor's type, name, performance parameters, and mission schedule. The manager periodically requests the experiment time from the Timer Manager; at times consistent with the mission schedule, the manager requests the details of the actual targets in the region under surveillance and filters the target data as a function of the weather conditions obtained from the Meteorological Data Manager and the sensor performance characteristics.

The simulated sensor data is stored in a file on the local host. This file preserves the sensor data, consisting of an image array of signal amplitudes, for a given sensor mission. Presently, due to the large amount of data stored, this file is accessed primarily by a local user for graphical display purposes. Remote users are given access to this file, using the Cronus COS directory interface; these interactions are mediated by Cronus access control mechanisms.

A more concise data format has been adopted for sharing the simulated sensor data with other processing components of the application. This format consists of a list of detections, each identifying a position whose sensed energy or temperature exceed a specified threshold for the particular region under surveillance. This list is an object which is collected and recorded for all of the sensor system instances by a common sensor data management service. Sensor data maintained in this format can be accessed remotely by any user authorized to do so. It is such data, collected from all sensor systems, that is used in the target data correlation process. However, this data can also be requested for display purposes.

As previously mentioned, the Cronus system model has been adopted for implementation of the sensor simulation and integration with other managers. Each sensor object stores the information necessary for the representation of an instance of a sensor system. The attributes of this object are described in Tables 4 and 5. Several operations can be performed on the sensor object. *Create* produces a new object describing an additional sensor system; *MoveTo* appends a mission to an existing sensor object; *Edit* can be used to modify the existing sensor description, including mission schedule; and *OperateSensor* is used to activate a sensor, starting the process of generating simulated sensor data.



Sensor Manager's Operation. *Operatesensor* Operation.  
Figure 11

Object Attributes	Cronus Type	Description
Type	SENSORTYPE	radar, infrared, or sigint
Name	ASC	sensor name
XYResolution	S16I	1 to 10 used presently (1->low resolution)
FrmDefs	array of FRMDEF	mission description

Sensor Object Attributes  
Table 4

Object Attributes	Cronus Type	Description
TStart	EDATE	starting time of surveillance mission
TEnd	EDATE	ending time of mission
TInc	EDATE	incremental time for scan of region
Location	LOC3D	location of platform performing surveillance
Region	REGION	coordinates describing rectangular region under surveillance

FRMDEF Attributes  
Table 5

The implementation described above has been influenced by the need to have multiple remote sensor systems supplying data to the ground target situation analysis component. A multiple sensor system configuration is obtained by operating the appropriate number of sensor software modules simultaneously; generally these instances are managed by different managers running on separate hosts to emulate real sensors existing as distinct physical resources.

As an example, a two sensor scenario was constructed. The sensor object descriptions are shown in Tables 6 and 7. A mission consists of moving the appropriate sensor to the specified *location* at time *tstart* and keeping it there until *tend*. At the end of each interval, specified by *tinc*, the sensor produces an image of the region bounded by *lowerleft* and *upperright*, saves the data and sends a detection report to the sensor data collector.

The target scenario consists of three targets whose true trajectories are shown in conjunction with a cartographic background in Figure 12, which is a photograph of a prototype display for C<sup>2</sup> Internet.

name: newrad  
type: radar  
xyresolution: 400  
frmdefs:  
    tstart: 0 minutes  
    tend: 50 minutes  
    tinc: 2 minutes  
location:  
    Latitude: N50 deg 30'0.0"  
    Longitude: E10 deg 0'0.0"  
    Altitude: 12000  
region:  
    lowerleft:  
        Latitude: N50 deg 36'0.0"  
        Longitude: E10 deg 24'0.0"  
        Altitude: 0  
    upperright:  
        Latitude: N51 deg 0'0.0"  
        Longitude: E11 deg 0'0.0"  
        Altitude: 0

Mission Schedule for Radar Sensor *newrad*  
Table 6

#### 5.2.4. Mission Data Manager

The Mission Data Manager records detection lists produced by the sensors during reconnaissance missions. Because this service must acquire data from a potentially large number of sensors, several managers collectively support the Mission Data Management service. A simple resource management algorithm is used to select which manager will be responsible for recording data acquired by a particular sensor, and mechanisms are included to redirect the recording function to another Mission Data Manager after a failure of the manager that was initially assigned.

The data for each mission is stored as a Cronus object, using the object database functions provided by the development tools. The attributes of this object are displayed in Tables 8 and 9.

Several operations can be performed on Mission Data objects. *StoreSensorData*, applied to the generic mission data object, creates a new mission data object and stores an initial detection frame with the object. The UID of the object is returned to the client; subsequent *StoreSensorData* operations should be applied directly to this newly created object. *GetSensorData* retrieves a set of detection lists. The client submits, with the request, the region which should be checked for detections, the time range during which the detections should have occurred, and the ID of the sensor that reported the detections; the time range and sensor identity are optional: if omitted, all entries for the given region will be reported. *GetMissionDetections* is similar to *GetSensorData*, except the search will be restricted to data recorded for a particular mission.

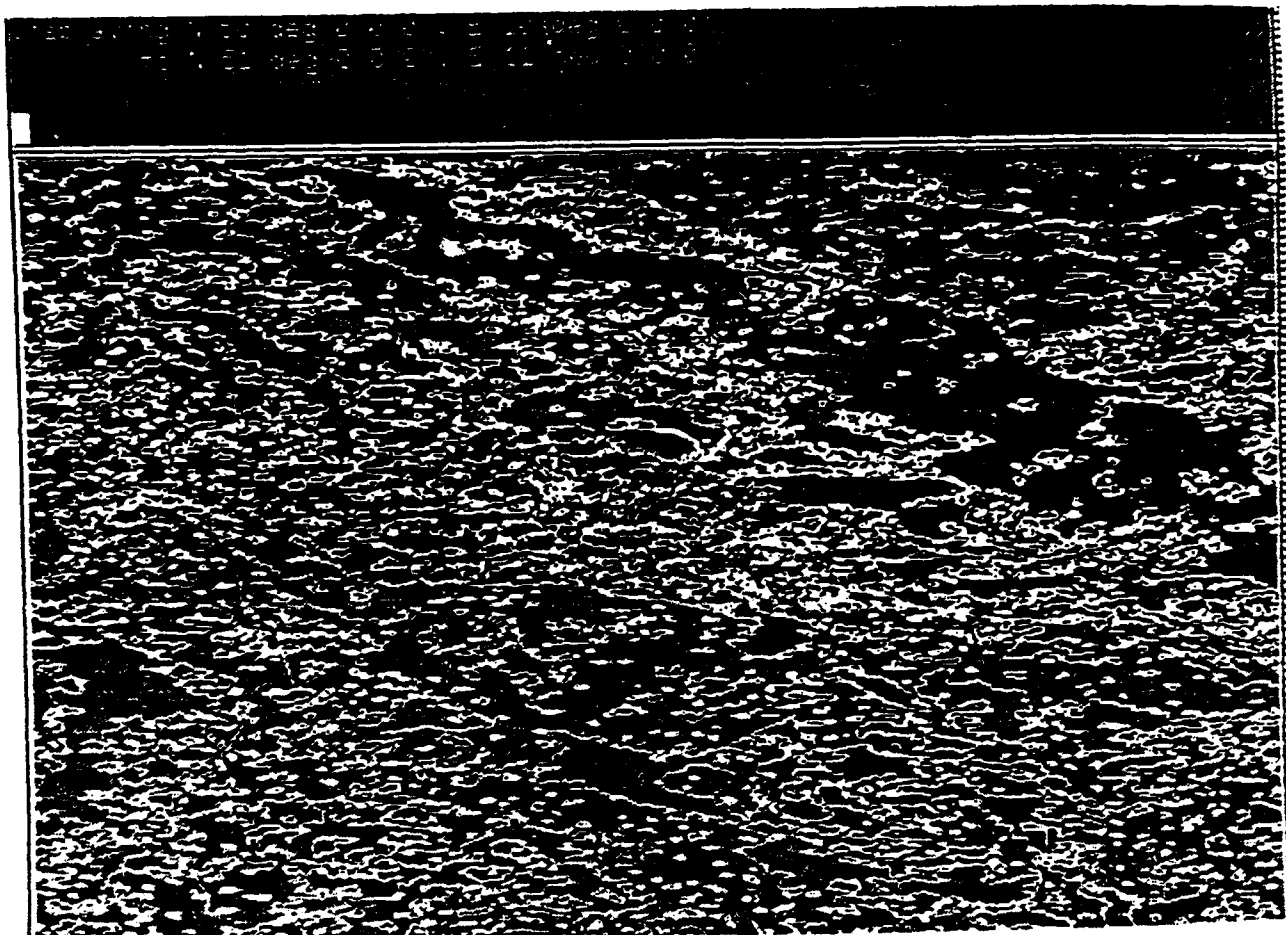
name: if3  
type: infrared  
xyresolution: 500  
frmdefs:  
  tstart: 10 minutes  
  tend: 45 minutes  
  tinc: 2 minutes  
  location:  
    Latitude: N50 deg 30'0.0"  
    Longitude: E10 deg 0'0.0"  
    Altitude: 12000  
  region:  
    lowerleft:  
      Latitude: N50 deg 33'0.0"  
      Longitude: E10 deg 24'0.0"  
      Altitude: 0  
    upperright:  
      Latitude: N50 deg 51'0.0"  
      Longitude: E10 deg 48'0.0"  
      Altitude: 0

Mission Schedule for Infrared Sensor *if3*  
Table 7

One additional feature of the Mission Data Manager is its use of resource management to select which site should be responsible for data reported during a particular mission. When the *StoreSensorData* for a new mission is first recieved by a manager, it selects which manager should be responsible by using a simple round-robin algorithm. In this algorithm, managers are assigned a sequence number when they are initially started. Missions are then assigned to managers in circular order. That is, if there are three managers, they would be numbered 1-2-3. The first mission goes to manager 1, the second to manager 2, the third to manager 3, and the fourth again to manager 1.

The selection is made by the manager that first recieved the request. The request is then forwarded to the selected manager using standard Cronus message forwarding mechanisms. The underlying Cronus support for object caches records the location of the new object so that additional requests to store data associated with the mission data object will be routed to the correct site.

These same basic mechanisms could be used with a more sophisticated selection algorithm. Such an algorithm might take account of proximity to the reporting sensor, the actual load at each recording site, or the reliability of each recording site. We chose a simple algorithm to expedite the initial implementation; a more complex algorithm might be introduced at a later time.



Example Target Trajectories  
Figure 12

Object Attributes	Cronus Type	Description
SensorID	EUID	sensor reporting the detections
Threshold	U16I	sensor's detection threshold for the mission
MissionNo	U32I	sensor's scheduled mission number for mission
FrameArray	array of DETECTIONFRAME	list of detections

Mission Data Attributes  
Table 8

Object Attributes	Cronus Type	Description
SampleTime	EDATE	date and time when detections where recorded
Detections	arrya of DETECTIONDATA	list of detections

DETECTIONFRAME Attributes  
Table 9

#### 5.2.5. Fusion (Detection Correlation) Processing

The fusion processing component takes collections of target events, detected during sensor missions, and performs various numeric and symbolic processing tasks to eliminate false alarms, correlate detections provided by different sensors, and identify target trajectories. Eventually, this function will be performed by a processor which is functionally specialized to the task of target correlation processing. In its initial version, the detection report fusion processing will be performed by a simple client program which is run manually by the operator.

The fusion algorithm itself is quite simple, and is intended for demonstration purposes only. Each detection submitted by the sensor manager is marked with the identity of the target which caused the event, or is marked as a false alarm resulting from noise. These markings are stored with the data in the Mission Data database, and are used by the fusion processor to determine the trajectory of each detected target. Entries which are false alarms are cast out; the remaining entries as sorted by target identity to collect together all entries for each target.

### 5.2.6. Target Report Manager

The Target Report objects each describe an object that has been detected by one of the sensing systems and verified by the fusion processing system. Each object may have additional data associated with it that describes attributes such as anticipated destination and priority.

Target reports are stored as Cronus objects, using the functions provided by the development tools.

Object Attributes	Cronus Type	Description
TDesc	TARGETDESC	summary of the target description
Sensors	array of SENSORDATA	detections associated with target

Target Report Attributes  
Table 10

Object Attributes	Cronus Type	Description
ReportUID	EUID	unique ID assigned to the report
TargUID	EUID	ID of the simulated target associated with report
Name	ASC	user chosen name associated with the target
Type	ENUM	kind of target, eg. tank, aircraft
Destination	LOC3D	known or anticipated destination
Priority	S32I	relative importance of the target
Weapon	ASC	weapon assigned to the target
Region	REGION	area in which the target has been detected
StartTime	EDATE	time the target was first detected
EndTime	EDATE	time the target was last detected

TARGETDESC Attributes  
Table 11

Each report has the attributes displayed in Tables 10, 11, 12, and 13.

Several operations can be performed on Target Report objects. *StoreTargetReport* creates a new target report. The initial description of the target is included in the request. *GetTargetDesc* returns a list target reports for all detected targets in a particular region. A time range may also be specified. *GetTargetReport* returns the description of particular target.

Object Attributes	Cronus Type	Description
Sensor	EUID	ID of sensor providing data
SensorName	ASC	name of sensor providing data
Detections	array of DETECTIONS	detections associated with target

SENSORDATA Attributes

Table 12

Object Attributes	Cronus Type	Description
SampleTime	EDATE	time at which target was sensed
Location	LOC3D	target's position at that time
snr	US2I	signal-to-noise ratio of the detection

DETECTIONS Attributes

Table 13

### 5.2.7. Meteorological Data Manager

The meteorological data manager provides a logically central source of weather report and forecast data, which can be used in the mission planning process. It is also used by the sensor simulator to imitate the effects of meteorological conditions on sensor performance.

To implement the Meteorological Data Manager, a set of weather attributes and appropriate canonical representations common to both reports and forecasts were chosen. These are maintained in the appropriate object structures shown in Tables 14 and 15. Note that weather reports and and weather forecasts are identical, except for their time-tagging information.

The operations which have been implemented to date include *Create* and *ShowWeatherNearLocation* (correspondingly, *Create* and *ShowForecastNearLocation*, for weather forecasts). The *Create* operation produces a weather report for a particular region as of a particular time. To process *ShowWeatherNearLocation*, the manager searches the weather report object database. For each report, it computes the distance between the requested location and the location tagging the report. The manager returns to the client the report for the location nearest to the location requested and includes the actual position represented in the report. This has the advantage that the client program does not need to specify the report by name or identify the precise location where the reporting station is located. Conversely, it has the disadvantage that the client might receive a report that is distant from the desired location, if none were from closer locations.

Object Attributes	Cronus Type	Description
ReportDate	EDATE	
Location	LOC	
Ceiling	U16I	in meters
CloudCoverage	CLOUDCOVERAGE	
CloudTypes	array of CLOUDTYPE	
Surface Visibility	U32I	in meters
VisibilityRestrictions	ASC	
Weather	WEATHERTYPE	
BarometricPressure	U16I	in millibars
Temperature	S16I	degrees Centigrade
DewPoint	S16I	degrees Centigrade
SurfaceWindDirection	U16I	in degrees, 0-360
AvgSurfaceWindSpeed	U16I	km per hour
MaxSurfaceWindSpeed	U16I	km per hour
Remarks	ASC	
Station	ASC	weather station name

Weather Report Object Attributes  
Table 14

The elimination of outdated weather reports is performed by the *DeleteOutdatedForecasts* operation. When the weather report is submitted, the meteorologist specifies an *EndValidityPeriod*. When *DeleteOutdatedForecasts* is invoked, each weather forecast object in the object database is examined and those whose validity period have expired are removed. Since the *DeleteOutdatedForecasts* affects processing on a collection of objects rather than on one specific object, this operation is implemented as an operation on the *generic object* (as is *ShowWeatherNearLocation* also). The implementation of the operation uses Cronus routines to scan all elements in the object database and to delete the objects that are out of date.

### 5.2.8. Cartographic Data Manager

Cartographic data is available in a variety of formats and from a variety of sources. We want to use available sources of cartographic data, and then provide a standard format for cartographic data used within the system. We also have to consider the speed with which data can be supplied by the manager due to the large volume of data needed to describe any given region. Both of these considerations are important in choosing a representation and in implementing the cartographic manager. Several sources and formats were reviewed prior to implementation and one was chosen: the Defense Mapping Agency's (DMA's) Digital Landmass System (DLMS) Data Base formats.

Attribute Name	Cronus Type	Description
ForecastDate	EDATE	
BeginValidityPeriod	EDATE	
EndValidityPeriod	EDATE	
Location	LOC	
Ceiling	U16I	in meters
CloudCoverage	CLOUDCOVERAGE	
CloudTypes	array of CLOUDTYPE	
SurfaceVisibility	U32I	in meters
VisibilityRestrictions	ASC	
Weather	WEATHERTYPE	
BarometricPressure	U16I	in millibars
Temperature	S16I	degrees Centigrade
DewPoint	S16I	degrees Centigrade
SurfaceWindDirection	U16I	in degrees, 0-360
AvgSurfaceWindSpeed	U16I	km per hour
MaxSurfaceWindSpeed	U16I	km per hour
Remarks	ASC	
Station	ASC	weather station name

Weather Forecast Object Attributes  
Table 15

The implementation of the manager will be built using Digital Feature Analysis Data (DFAD) available from DMA. This data provides, in a standardized format, a description of the planimetric features in a given geographic region. For each feature contained in a DFAD description, there is a *feature type* identifying the feature as an area, line, or point, a *feature identification code* describing the feature (e.g., forest, road, industry), and a list of latitude and longitude coordinates specifying the location of the feature. Using this basic information, a manager implementing the desired operations can be constructed.

In order to improve performance, the manager will rely on both the local filesystem and the object database for storage. Since parsing a DFAD is a processor-intensive operation, it is desirable to minimize the delay when retrieving feature information for a client. In order to decrease this retrieval delay, the manager will use the object database as a cache for the (less compact but more detailed) information stored in the file. The local filesystem will be used to store the complete DFAD for a particular area (a map). Creating a new feature map object from the data provided from DMA will involve a number of steps, including the creation of the appropriate cache information.

1. The operator loads the raw DMA feature data into a local file;
2. The operator then creates a feature map object that corresponds to the feature data file using the operations to

- Parse the DFAD file; and
- Create the cache by recording an object database entry for the newly created object which describes the geographic area covered by the file, and which includes a sorted list of the identification codes of features in the file.

The *ShowFeaturesInArea* operation retrieves classes of features for a given region. The operation involves the following process.

1. Look at a feature map's cache in the object database.
2. Based on the cached information, does this feature map cover the correct geographic region and contain the features requested?
3. If so, reparse the corresponding local file containing the raw feature data and return the appropriate features; otherwise continue searching the object database.

Object Attributes	Cronus Type	Description
Name	ASC	map name
Host	EHOST	host on which the map resides
COSPath	ASC	file in which the map resides
SWCorner	LOC	SW corner of the map object
NECorner	LOC	NE corner of the map object
FeatureIdList	array of U32I	cache of features contained in the map

Feature Map Object Attributes  
Table 16

The information characterizing a feature map can be found in Table 16.

Since retrieving information from the cartographic data manager can potentially return a large quantity of data, the implementation will use Cronus large message support to return the requested information. This also has performance benefits, since the overhead associated with initiating the data transfer with the client is incurred only once, rather than once for each of several small messages needed to pass the data.

Since the total volume of cartographic data is quite large, it might prove inefficient to store it in one cluster and frequently transmit it to the other on each request. Therefore, the cartographic data will be replicated and exist in both clusters. This replication also improves survivability since requests from members of one cluster could automatically be processed by the cartographic data manager of the other cluster, should one of the cartographic data managers fail.

The particular data we are now using contains such features as power generation and distribution facilities, wooded areas and clearings, settlements, agricultural facilities, and significant surface water such as streams and ponds. A sample of the cartographic information is shown in Figure 13, which is a photograph of a prototype display for C<sup>2</sup> Internet.

#### 5.2.9. Resources and Logistics Data Management System

The Resources and Logistics Data Management System will ultimately provide capability and availability data for a variety of command and control resources. At present, the resources and logistics data manager is a stubbed component which supports a minimal facility for managing capability data for the vehicle object type. The *List Vehicles With Capability* operation performs a database-like function: it allows a client to specify a number of vehicle attributes and returns the names of vehicles having those characteristics. This operation is currently implemented by having the manager scan the objects in its object database and perform the appropriate matching. This type of processing seems well suited for implementation using a traditional database system: this is a possible future direction for the development of this component.

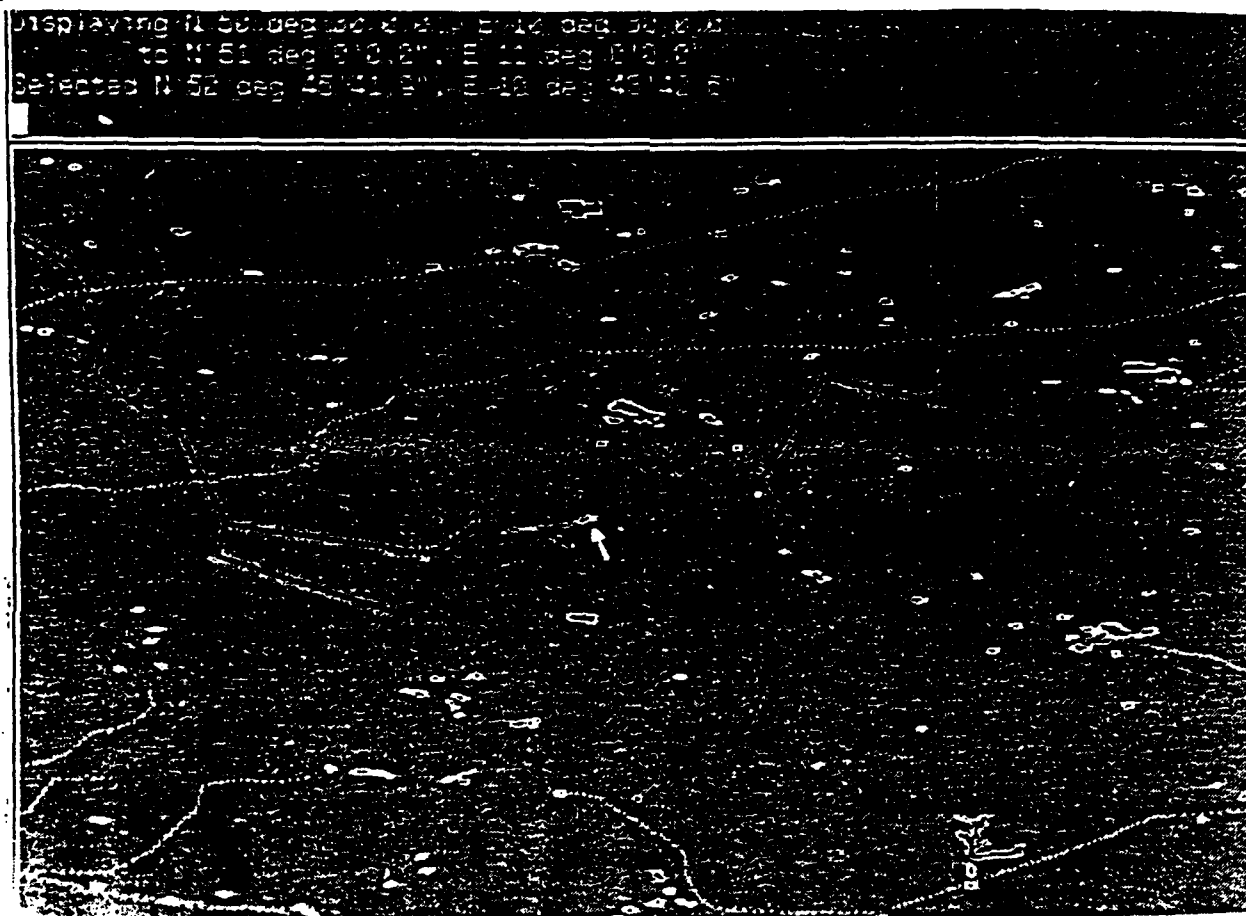
Object Attributes	Cronus Type	Description
VehicleName	ASC	e.g., "A-10A"
VehicleType	ASC	e.g., "single-seat close support aircraft"
MaximumSpeed	U16I	kilometers per hour
MaximumWeight	U16I	kilograms at takeoff
MaximumRange	U16I	kilometers without refueling
MaximumFuel	U16I	liters
TakeoffLength	U16I	meters
LandingLength	U16I	meters

Vehicle Object Attributes  
Table 17

The attributes of a vehicle object are shown in Table 17.

#### 5.2.10. Application Monitoring and Control

The MCS for the C<sup>2</sup> application must be capable of acquiring status information about application components at sites throughout the cluster, and be capable of organizing the display of that information so that it can be easily used by an operator. Control of the application components must be integrated with the monitoring function, since component status often affects operator decisions. The operator may regulate the monitoring activities of the MCS, by changing the focus of attention as monitoring needs change.



Cartographic Display Image with Target Tracks  
 Figure 13

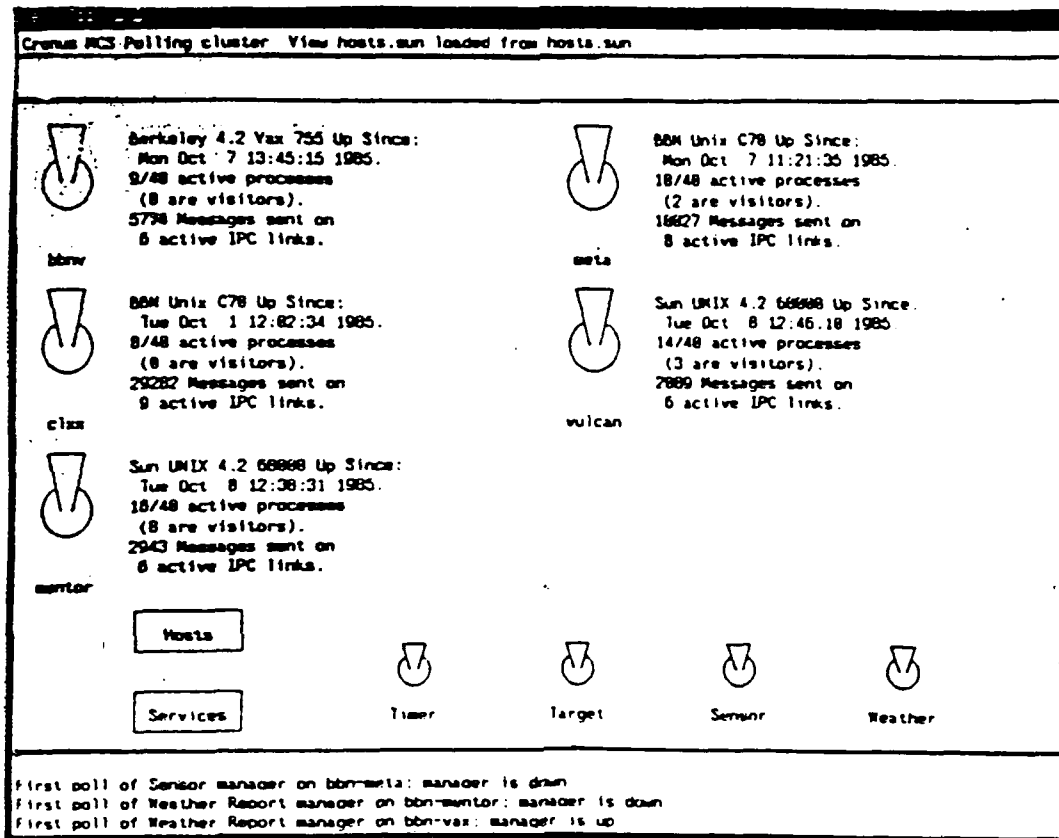
We built the monitoring and control facilities for the C<sup>2</sup> Internet application using the facilities of the existing Cronus MCS, which provides integrated monitoring and control of the underlying Cronus operating system and the hosts on which it operates. The Cronus MCS was extended by identifying the application components that were to be monitored and describing how their status information should be displayed.

One view of the system, Figure 14, is a photograph of the MCS showing several hosts that support the C<sup>2</sup> application. Each host is represented by a toggle switch with the name of the host underneath; the up position of the switch indicates that the host is available; the down position of the switch indicates that the host is unavailable. Each available host has an associated status summary. For example, we see that host bbnv is available; it has been running since Oct 7, 1985, has 9 Cronus processes running, and has sent 5734 messages to 6 other hosts since it was started. The view also displays a summary of selected services at the bottom of the screen. In this particular view, the status of the timer, target, sensor and weather services are shown; all of those services are available.

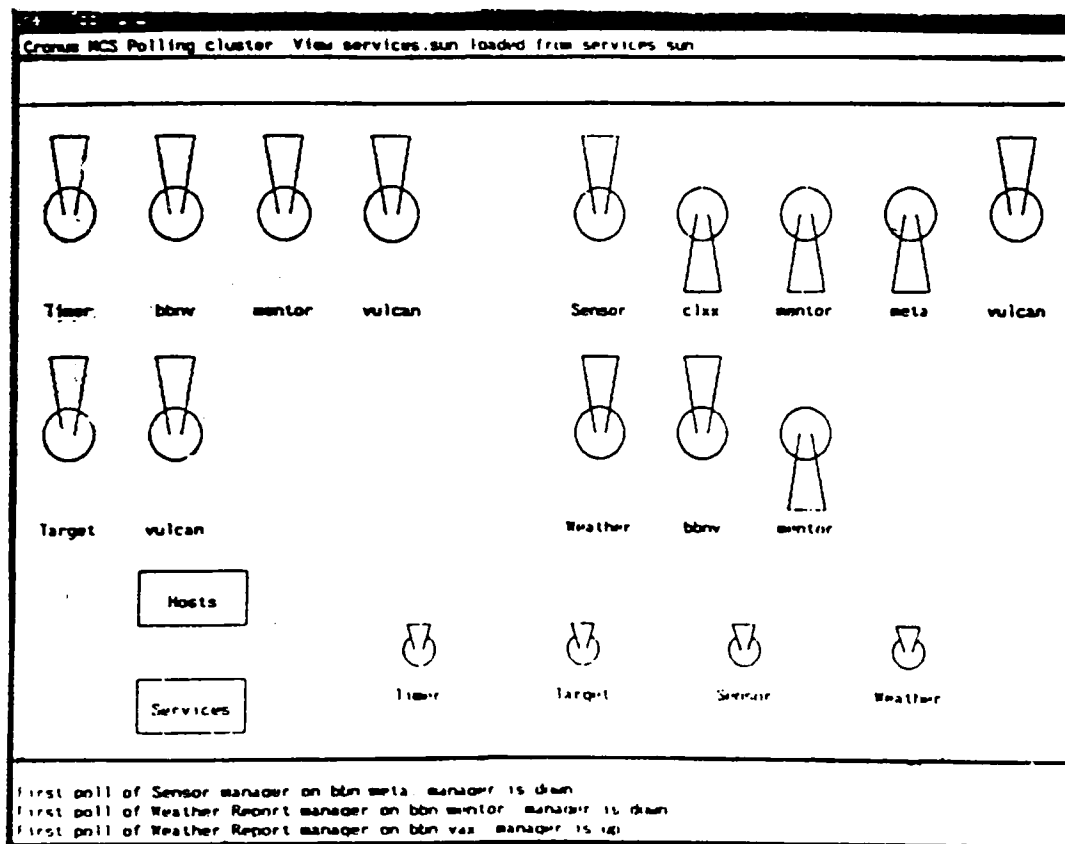
The control of the hosts and managers monitored by the MCS is integrated with the status display. For example, Cronus operation on a host can be suspended or resumed through the MCS by selecting the icon that represents the component and then choosing either *suspend* or *resume* from a menu that the MCS displays. Individual managers can be started or stopped in a similar way. This type of interaction with the MCS may also be used to adjust how often status data is requested from various managers and to change the status information that is displayed.

A view of the status of the services is shown in Figure 15, another photograph of the MCS display. The managers for each type are grouped together, with a summary of their status displayed at the left of the grouping. For the timer manager, we see that the service is available, supported by three operating managers. The sensor service is also available, but only one of the four instances of the manager is currently active. The target service is supported by only a single manager (which is active), and the weather service is supported by two managers (one of which is active).

The MCS also will notify the operator when irregular events occur, using a window at the bottom of the display. When event reports from managers arrive, or when the MCS notices an important change in status, a message will be displayed in the this lower window. In the examples we have given, reports about the availability of the sensor manager and the two instances of the weather manager are visible.



Application MCS Host Status Display  
Figure 14



Application MCS Service Status Display  
 Figure 15

## 6. SUMMARY AND CONCLUSIONS

To date we have designed and implemented a number of  $C^2$  Internet functional components in accordance with the Cronus system architecture and using the Cronus application development tools. This activity has so far been highly successful. It has resulted in working, demonstrable, integrated prototypes of a number of parts of the application with relatively little manpower applied to the development and integration tasks. It has also resulted in initiating a number of improvements to both the Cronus system and its support tools based on the initial design and implementation experience. Finally, it has substantiated the already anticipated need for the introduction and refinement of supplemental technologies within the Cronus context in order to extend the domain of integrated resources available for application use.

In follow-on phases we anticipate incorporating additional technology areas such as flexible multi-cluster support, integration of database functionality, support for symbolic processing systems to experiment with "expert systems" substituting for simple stubbed programs or user interaction in current scenarios, and integration of high performance processing elements for compute-intensive application functions. Obviously, a prerequisite to evolving the  $C^2$  Internet scenarios to incorporate these technologies is the integration of the raw technology base into the Cronus environment.

## 7. REFERENCES

- ADDCOMPE83. B. Leiner, T. Klein, and M. Frankel, *Army/DARPA Distributed Communications/Processing Experiment Technical Plan*. November 1983.
- BBN-5879. Richard E. Schantz and Robert H. Thomas, *Cronus, A Distributed Operating System: Functional Definition and System Concept*, BBN Laboratories, Technical Report 5879 (June 1982, Revised January 1985).
- BBN-5884. Richard E. Schantz and Robert H. Thomas, *Cronus, A Distributed Operating System: System/Subsystem Specification*, BBN Laboratories, Technical Report 5884 (June 1982, Revised February 1986).
- BBN-5942. J. Berets, R. Mucci, R. Schantz, and K. Schroder, *C2 System Internet Experiment: Functional Definition*, BBN Laboratories, Technical Report No. 5942 (May 1985, Revised May 1986).
- BBN-6073. J. Berets, R. Mucci, R. Schantz, and K. Schroder, *C2 System Internet Experiment: Interim Technical Report*, BBN Laboratories, Technical Report No. 6073 (October 1985).
- BBN-6180. R. Sands and K. Schroder, eds., *Cronus User's Manual*, BBN Laboratories, Technical Report 6180 (February 1986).
- BBN-6181. R. Sands and K. Schroder, eds., *Cronus Program Maintenance Manual*, BBN Laboratories, Technical Report 6181 (February 1986).
- BBN-6251. J. Berets, R. Mucci, R. Schantz, and K. Schroder, *C2 System Internet Experiment: Final Technical Report*, BBN Laboratories, Technical Report No. 6251 (May 1986).
- Dean86. Michael A. Dean, Richard M. Sands, and Richard E. Schantz, *Canonical Data Representation in the Cronus Distributed Operating System*, BBN, unpublished paper (February 1986).
- Gurwitz86. R. Gurwitz, M. Dean, and R. Schantz, *Programming Support in the Cronus Distributed Operating System*, Sixth International Conference on Distributed Computing Systems (May 1986).
- Schantz86. R. Schantz, R. Thomas, and G. Bono, *The Architecture of the Cronus Distributed Operating System*, Sixth International Conference on Distributed Computing Systems (May 1986).
- TRW80. M. Mariani, R. Turn, and B. Johnson, "Architectural Study for Tactical C<sup>3</sup> Construct," *RADC Technical Report 80-304*, (September 1980).

## A. APPENDIX: APPLICATION COMPONENT DOCUMENTATION

The articles in this section describe the interfaces and canonical types for invoking requests on objects supported by each of the application services that have been developed for the C<sup>2</sup> Internet Experiment application. The information in these articles is maintained along with other descriptive information kept by the Cronus type definition manager; the formatting of the articles is performed automatically by the tools to the type definition manager.

**Cronus type:** CT\_Airfield (subtype of CT\_C2Internet\_Object)

The CT\_Airfield object contains information describing an airfield: its location and runway configuration.

**Generic operation:** Create

An airfield object is created and the supplied description of it is stored. The airfield is cataloged in the Cronus directory :user:c2inet:airfields with the specified name.

This operation requires generic create rights.

```
Create(Airfield: AIRFIELDINFO;  
      {ObjectUID: EUID};  
      {IACLHints: array of EUID})  
=> (ObjectUID: EUID)
```

**Airfield**

is the description of the airfield.

**ObjectUID**

is an optional specification of the UID to be assigned to the newly created object.

**IACLHints**

provides optional hints for initializing the Access Control List for the new object. A detailed description may be found in the annotations for the Create operation on type CT\_Object.

**ObjectUID**

is the UID of the newly created object.

**Generic operation:** FindNearestAirfield

The FindNearestAirfield operation returns the airfield closest to the specified location.

This operation requires generic search rights.

```
FindNearestAirfield(Where: LOC)  
=> (Airfield: AIRFIELDINFO)
```

**Where**

is the location for which the nearest airfield is being requested.

**Airfield**

is the description of the nearest airfield.

**Operation:** DescribeAirfield

The DescribeAirfield operation returns a description of the specified airfield

This operation requires showfield rights.

**DescribeAirfield()**

=> (Airfield: AIRFIELDINFO)

**Airfield**

is a the description of the requested airfield.

**Operation: ListRunways**

The ListRunways operation returns the runways of a particular airfield.

This operation requires showfield rights.

**ListRunways()**

=> (RunwayList: array of RUNWAY)

**RunwayList**

is a description of the runways at the requested airfield.

*Canonical Types*

**Cantype: SURTYPE**

A SURTYPE is one of the possible runway surface types.

**SURTYPE: { GRASS, CONCRETE, ASPHALT };**

**GRASS**

indicates a grass runway.

**CONCRETE**

indicates a concrete runway.

**ASPHALT**

indicates an asphalt runway.

**Cantype: RUNWAY**

The RUNWAY cantype describes an airfield's runway.

**RUNWAY: record**

Length: U16I;

Bearing: U16I;

Surface: SURTYPE;

end;

**Length**

is the runway length in meters.

**Bearing**

is the runway bearing 0 - 360 degrees.

**Surface**

is the runway surface type.

**Cantype: AIRFIELDINFO**

The AIRFIELDINFO cantype describes a complete airfield.

**AIRFIELDINFO: record**

AirfieldName: ASC;

Where: LOC3D;

RunwayList: array of RUNWAY;

NavAidList: ASC;

end;

**AirfieldName**

is the name of the airfield.

**Where**

is the airfield's location.

**RunwayList**

is a list of the configuration of the airfield's available runways.

**NavAidList**

is a textual description of any available navigational aids.

**Cronus type:** CT\_C2Internet\_Object (subtype of CT\_Object)

The C2Internet\_Object type defines those canonical types which are shared by more than one of the C2 Internet managers.

### *Canonical Types*

**Cantype:** LOC

The LOC cantype represents a location on the earth using latitude and longitude coordinates. Each of these is represented in integer tenths of seconds. North latitude and east longitude are represented as positive quantities, while south latitude and west longitude are represented as negative ones.

LOC: record

Latitude: S32I;  
Longitude: S32I;  
end;

**Latitude**

is the north / south position.

**Longitude**

is the east / west position.

**Cantype:** LOC3D

The LOC3D cantype represents a location on the earth using latitude, longitude, and altitude coordinates. Latitude and longitude are represented in integer tenths of seconds. North latitude and east longitude are represented as positive quantities, while south latitude and west longitude are represented as negative ones. Altitude is in meters above sea level.

LOC3D: record

Latitude: S32I;  
Longitude: S32I;  
Altitude: S32I;  
end;

**Latitude**

is the north / south position.

**Longitude**

is the east / west position.

**Altitude**

is the height in meters above sea level.

**Cantype:** REGION

A REGION specifies a rectangular volume using the locations specified by its two corners.

REGION: record  
  lowerleft: LOC3D;  
  upperright: LOC3D;  
  end;

lowerleft  
  is the southwest corner of the region.

upperright  
  is the northeast corner of the region.

Cantype: FRMDEF

A FRMDEF stores one element (or frame) of a sensor's schedule. Each FRMDEF is equivalent to a sensor mission.

FRMDEF: record  
  tstart: EDATE;  
  tend: EDATE;  
  tinc: EDATE;  
  location: LOC3D;  
  region: REGION;  
  end;

tstart  
  is the time at which the sensor will begin observing the specified area.

tend  
  is the time at which the sensor will finish observing the specified area.

tinc  
  is the interval the sensor will pause between subsequent observations.

location  
  is the sensor location during the mission.

region  
  is the region of surveillance for the sensor.

Cantype: SENSORTYPE

A SENSORTYPE denotes a class of similar remote detectors.

SENSORTYPE: { radar, infrared, sigint };

radar  
  is a system for locating an object by means of ultrahigh-frequency radio waves reflected from the object and received, observed, and analyzed such that characteristics of the object may be determined.

**infrared**

is a system for locating an object by means of thermally sensing and analyzing the spectrum emitted by the object such that characteristics of the object may be determined.

**sigint**

is a system for locating an object by means of intercepting and analyzing emitted radio signals such that the characteristics of the object may be determined.

**Cantype: SENSORVARIABLE**

A SENSORVARIABLE stores the complete description of a sensor, and consists of the sensor's name, type, resolution, and a sequence of missions.

**SENSORVARIABLE: record**

```
name: ASC;
type: SENSORTYPE;
xyresolution: S16I;
frmdefs: array of FRMDEF;
end;
```

**name**

is the sensor's name.

**type**

is the kind of sensor.

**xyresolution**

is the resolution performance parameter for the sensor (in deciseconds).

**frmdefs**

is the sensor's schedule.

**Cantype: DETECTIONDATA**

The DETECTIONDATA canonical type represents a target that has been detected by a sensor.

**DETECTIONDATA: record**

```
location: LOC3D;
targetuid: EUID;
snr: U16I;
end;
```

**location**

is the location of the detected event.

**targetuid**

is the UID of the target.

**snr**

is the signal-to-noise ratio of the detection.

**Cantype: SEGMENT**

The SEGMENT cantype is a parametric description of one segment of a target track trajectory.

**SEGMENT: record**

starttime: EDATE;  
startloc: LOC3D;  
endtime: EDATE;  
endloc: LOC3D;  
end;

**starttime**

is the time at which the target will begin this track segment.

**startloc**

is the initial position of the target.

**endtime**

is the time at which the target will complete this track segment.

**endloc**

is the final position of the target.

**Cantype: TARGETTYPE**

The TARGETTYPE cantype describes various kinds of objects of interest to the sensing systems.

TARGETTYPE: { jeep, tank, footsoldier, airplane, building, airfield, truck, bridge, powerplant, railroad, dam };

**jeep**

is a small general-purpose motor vehicle.

**tank**

is an enclosed, heavily armed and armored combat vehicle that moves on two endless metal belts.

**footsoldier**

is an infantryman.

**airplane**

is a fixed-wing heavier-than-air vehicle driven by a propeller or jet.

**building**

is a roofed and walled structure built for permanent use.

**airfield**

is a field maintained for the landing and takeoff of aircraft and for receiving and discharging passengers and cargo.

**truck**

is a wheeled vehicle for moving heavy articles.

**bridge**

is a structure carrying a pathway or roadway over a depression or obstacle.

**powerplant**

is an electric utility generating station.

**railroad**

is a line of track providing a path for cars or equipment drawn by locomotives or propelled by self-contained motors.

**dam**

is a barrier preventing the flow of water or of loose solid materials.

**Cantype: TARGETVARIABLE**

The TARGETVARIABLE cantype contains the complete description of a simulated target.

**TARGETVARIABLE: record**

name: ASC;  
targtype: TARGETTYPE;  
segments: array of SEGMENT;  
end;

**name**

is the name of the target.

**targtype**

is the kind of target.

**segments**

is a description of the target's path.

**Cronus type:** CT\_Feature\_Map (subtype of CT\_C2Internet\_Object)

The Feature Map Manager maintains and facilitates retrieval of cartographic information. A CT\_Feature\_Map object consists of two parts. The primary object is a file in the host's local filesystem that contains Digital Feature Analysis Data in the format specified by the Defense Mapping Agency (Product Specifications for Digital Landmass System Data Base DMA stock #SPEXDLMS2). The manager is capable of parsing data in this format and providing it remotely to client programs. For each feature file, corresponding summary information is kept in the Feature Map Manager's object database. The object database is used as a caching mechanism to enhance the retrieval speed of the cartographic information.

**Generic operation: Create**

The create operation attempts to create a MAPIDRECORD for the file on the specified Host and with the specified COSPath. If the manager is able to open the file and parse through the first feature in the file, success is assumed for the remainder of the file and create returns. The manager will continue to parse the file and any errors occurring subsequently will be recorded in the manager's log file.

This operation requires generic create rights.

```
Create(Name: ASC;  
      Host: EHOST;  
      COSPath: ASC;  
      [ObjectUID: EUID];  
      [IACLHints: array of EUID])  
=> (SWCorner: LOC;  
    NECorner: LOC)
```

**Name**  
is the name assigned to the map.

**Host**  
is the host on which the map resides.

**COSPath**  
is the local file in which the map will be found.

**ObjectUID**  
is an optional specification of the UID to be assigned to the newly created object.

**IACLHints**  
provides optional hints for initializing the Access Control List for the new object. A detailed description may be found in the annotations for the Create operation on type CT\_Object.

**SWCorner**  
is the latitude and longitude of the southwestern corner of the map.

**NECorner**  
is the latitude and longitude of the northeastern corner of the map.

**Generic operation: ShowFeaturesInArea**

The ShowFeaturesInArea operation will return all features contained in the region specified by SWCorner and NECorner whose FeatureId's match those requested. Omitting the FeatureIdList on the request returns all features for the specified region. Varying degrees of granularity may be obtained by requesting categories of FeatureId's. For each FeatureId in the FeatureIdList specified as input parameter, the following algorithm for returning features is used:

x00 returns all features with FeatureId xnn (general category); xy0 returns all features with FeatureId xyn (specific category); xyz returns features with FeatureId xyz only.

For example, specifying a FeatureIdList of 200, 403, 510 would return all features with FeatureId's 200-299, 403, or 510-519.

It is important to note that the Feature Map Manager is currently implemented only for the 32 bit architecture, byte-within-word and bit-within-byte ordering used on the SUN workstation. To increase efficiency, the manager does not do internal to canonical conversion prior to returning data from the ShowFeaturesInArea operation. A description of the client interface to be used when invoking this operation may be found in the C2 Internet Program Maintenance Manual.

This operation requires generic search rights.

```
ShowFeaturesInArea(SWCorner: LOC;
                   NECorner: LOC;
                   [FeatureIdList: array of U32I])
=> ()
```

**SWCorner**

is the southwest corner of the region for which feature data is wanted.

**NECorner**

is the northeast corner of the region for which feature data is wanted.

**FeatureIdList**

is a list of the kinds of features which are wanted.

**Generic operation: ReportStatus**

The ReportStatus operation dumps a collection of statistics that are accumulated as the Feature Map Manager runs.

This operation requires generic status rights.

```
ReportStatus()
=> (Bootdate: EDATE;
    OpCount: U32I;
    OctetsSent: U32I;
    UserCPUSeconds: U32I;
    SystemCPUSeconds: U32I;
    AvgOctetsPerOp: U32I;
    AvgUserCPUSecondsPerOp: U32I;
```

AvgOctetsPerUserCPUSec: U32I;  
MgrSize: U16I)

**Bootdate**

is the date and time at which the manager was most recently started.

**OpCount**

is the number of operations that the manager has processed since it was started.

**OctetsSent**

is the number of octets the manager has sent out as feature data returned as responses to the ShowFeaturesInArea operation. This number does not include octets sent as the result of other operations.

**UserCPUSeconds**

is the number of seconds of user processor time the manager has used since it was started.

**SystemCPUSeconds**

is the number of seconds of system processor time the manager has used since it was started.

**AvgOctetsPerOp**

is the ratio of the number of octets the manager has returned as responses to the ShowFeaturesInArea operation to the total number of operations completed.

**AvgUserCPUSecondsPerOp**

is the ratio of the number of seconds of user processor time the manager has used since it was started to the total number of operations it has completed.

**AvgOctetsPerUserCPUSeconds**

is the ratio of the number of octets the manager has returned as responses to the ShowFeaturesInArea operation to the total number of seconds of user processor time the manager has used since it was started.

**MgrSize**

is the maximum resident set size of the manager in kilobytes.

**Canonical Types****Cantype: MAPIDRECORD**

A MAPIDRECORD is the representation used to cache information about a feature map file in the manager's object database.

**MAPIDRECORD: record**

Name: ASC;  
Host: EHOST;  
COSPath: ASC;  
SWCorner: LOC;  
NECorner: LOC;

FeatureIdList: array of U32I;  
end;

**Name**

is the name assigned to the map.

**Host**

is the host on which the map resides.

**COSPath**

is the local file in which the map will be found.

**SWCorner**

is the latitude and longitude of the southwestern corner of the map.

**NECorner**

is the latitude and longitude of the northeastern corner of the map.

**FeatureIdList**

summarizes the kinds of features to be found in the file. Each feature is assigned a FeatureId that classifies it into a well-known category (for example, lake, power plant, agricultural building). Each element in the array indicates the number of features of each kind to be found in the feature file.

**Cronus type:** CT\_Mission\_Data (subtype of CT\_C2Internet\_Object)

The Mission Data Manager is responsible for the long-term storage of targets detected by the Sensor Managers. Each sensor is chronologically scheduled to perform one or more missions. When a sensor runs a mission, the resulting data is stored as one complete Mission\_Data object (in the absence of failures) made up of one or more DETECTIONFRAMEs. A Sensor Manager stores mission data by initially invoking the StoreSensorData operation on the generic Mission\_Data object. This, in turn, causes the Mission Data Manager to perform resource management, and either accept the sensor's request, or redirect it to an alternate Mission Data Manager. Subsequent storage of sensor data for the same mission is achieved when the Sensor Manager invokes the (non-generic) StoreSensorData operation on the UID returned by the initial (generic) invoke. Clients may retrieve data from the Mission Data Manager by invoking the GetSensorData operation and specifying any desired parameters. The responding Mission Data Manager uses the GetMissionDetection operation to collect the relevant information from its peer managers, aggregates this information, and finally returns the results to the requesting client.

**Generic operation:** StoreSensorData

The generic StoreSensorData operation stores a set of detections with a Detection Data Manager.

```
StoreSensorData(SensorID: EUID;
                SampleTime: EDATE;
                Threshold: U16I;
                MissionNo: U32I;
                [SensorVar: SENSORVARIABLE];
                Detections: array of DETECTIONDATA)
=> (MissionID: EUID)
```

**SensorID**

is the UID of the sensor attempting to store the data.

**SampleTime**

is the date and time when this data set was recorded by the sensor.

**Threshold**

is the threshold that the sensor has used to discern events from noise in this data set.

**MissionNo**

is the mission for the specified sensor for which detections are being stored.

**SensorVar**

is a description of the sensor.

**Detections**

is a list of the detections in this data set.

**MissionID**

is the UID of the mission being stored.

**Operation:** StoreSensorData

The StoreSensorData operation stores a set of detections with a specified Detection Data

**Manager.**

**StoreSensorData**(SensorID: EUID;  
SampleTime: EDATE;  
Threshold: U16I;  
MissionNo: U32I;  
[SensorVar: SENSORVARIABLE];  
Detections: array of DETECTIONDATA)  
=> (MissionID: EUID)

**SensorID**

is the UID of the sensor attempting to store the data.

**SampleTime**

is the date and time when this data set was recorded by the sensor.

**Threshold**

is the threshold that the sensor has used to discern events from noise in this data set.

**MissionNo**

is the mission for the specified sensor for which detections are being stored.

**SensorVar**

is a description of the sensor.

**Detections**

is a list of the detections in this data set.

**MissionID**

is the UID of the mission being stored.

**Operation: SetObjectiveVar**

The SetObjectiveVar operation allows an operator to set a variable that will control the behavior of the manager when the resource management algorithm is executed.

**SetObjectiveVar**(ObjectiveVar: U32I)  
=> ()

**Objective Var**

is the variable.

**Generic operation: GetMgrStatus**

The GetMgrStatus operation retrieves resource management information from a Mission Data Manager's peers.

**GetMgrStatus**()  
=> (ObjectiveVar: U32I;  
LastMission: U32I)

**Objective Var**

is an operator-controllable variable that affects the behavior of the manager during resource management.

**LastMission**

is the number of seconds since the manager last began storing a mission.

**Generic operation: GetSensorData**

The GetSensorData operation aggregates sensor information from a collection of Mission Data Managers, and returns it to the requesting client.

```
GetSensorData(Region: REGION;  
  [StartTime: EDATE];  
  [EndTime: EDATE];  
  [SensorID: EUID])  
=> ([Detections: array of DETECTIONEVENT])
```

**Region**

is the area for which sensor information is requested.

**StartTime**

is the beginning of the time window in which the client is interested.

**EndTime**

is the end of the time window in which the client is interested.

**SensorID**

is the UID of the sensor for which data is being requested. Omitting the SensorID retrieves data from all available sensors.

**Detections**

is a list of the detections meeting the specified requirements.

**Operation: GetMissionDetection**

The GetMissionDetection operation retrieves any detections for the specified mission, region, time window, and sensor.

```
GetMissionDetection(MissionID: EUID;  
  Region: REGION;  
  [StartTime: EDATE];  
  [EndTime: EDATE];  
  [SensorID: EUID])  
=> ([Detections: array of DETECTIONEVENT])
```

**MissionID**

is the mission from which detections will be retrieved.

**Region**

is the area within which detections will be retrieved.

**StartTime**

is the beginning of the time window in which the client is interested.

**EndTime**

is the end of the time window in which the client is interested.

**SensorID**

is the UID of the sensor for which the mission was created.

**Detections**

is a list of the detections meeting the specified requirements.

**Operation: Remove**

The Remove operation removes the specified Mission\_Data object.

This operation requires remove rights.

**Remove()**

=> ()

*Canonical Types***Cantype: DETECTIONFRAME**

The DETECTIONFRAME cantype is a time-tagged list of events detected by a sensor.

**DETECTIONFRAME: record**

SampleTime: EDATE;  
Detections: array of DETECTIONDATA;  
end;

**SampleTime**

is the date and time when the detections were recorded.

**Detections**

is a list of detections.

**Cantype: MISSION**

The MISSION cantype details the detections found during a sensor mission.

**MISSION: record**

SensorID: EUID;  
Threshold: U16I;  
MissionNo: U32I;  
FrameArray: array of DETECTIONFRAME;  
end;

**SensorID**

is the UID of the sensor reporting the detections.

**Threshold**

is the sensor's threshold for the mission.

**MissionNo**

is the mission for the specified sensor during which the detections were reported.

**FrameArray**

is the list of the detections reported.

**Cantype: DETECTIONEVENT**

The DETECTIONEVENT cantype describes a single detection made by a sensor.

**DETECTIONEVENT: record**

time: EDATE;  
sensor: EUID;  
detection: DETECTIONDATA;  
end;

**time**

is the date and time when the detection were recorded.

**sensor**

is the UID of the sensor that recorded the detection.

**detection**

is a description of the event reported by the sensor.

**Cronus type:** CT\_Sensor (subtype of CT\_C2Internet\_Object)

A Sensor Manager mediates access to one or more CT\_Sensor objects. A CT\_Sensor object represents a real, physical sensor system. The Sensor Manager emulates the operation of these sensor systems by obtaining true target state information from the Target Simulation Manager and altering it in a manner consistent with the specified sensor performance parameters. Each CT\_Sensor may be chronologically scheduled for one or more missions. These missions specify where and when particular sensors should be performing observations. The conglomeration of the assigned missions for a sensor is known as the sensor's schedule. Each Sensor Manager is capable of simulating at most one physical sensor at a time. That is, a Sensor Manager cannot run more than one mission simultaneously, either for different or the same CT\_Sensor objects.

### *Sensor Configuration and Scheduling*

#### **Generic operation: Create**

The Create operation defines a new sensor.

```
Create(Name: ASC;  
      SensorType: SENSORTYPE;  
      Resolution: S32I)  
=> (ObjectUID: EUID)
```

#### **Name**

is the sensor's name.

#### **SensorType**

is the kind of sensor (radar, infrared, or signal interception).

#### **Resolution**

is the resolution of the sensor in deciseconds. This value will determine the geographical precision with which the sensor is capable of detecting targets. At the equator, one decisecond is equal to approximately three meters. Hence values in the hundreds of deciseconds are usually appropriate.

#### **ObjectUID**

is the UID of the newly created sensor object.

#### **Operation: MoveTo**

The MoveTo operation adds a mission descriptor to a sensor's schedule.

```
MoveTo(StartTime: EDATE;  
      EndTime: EDATE;  
      TimeInc: EDATE;  
      SurRegion: REGION;  
      SenLocation: LOC3D)  
=> ()
```

#### **StartTime**

is the time at which observations of the specified region will commence.

**EndTime**

is the time at which observations of the specified region will cease.

**TimeInc**

is the time increment between successive observations of the specified region.

**SurRegion**

is the region to be observed.

**SenLocation**

is the location of the sensor.

**Operation: ShowSchedule**

The ShowSchedule operation retrieves the schedule for the specified sensor.

ShowSchedule()

=> (SenSchedule: SENSORVARIABLE)

**SenSchedule**

is the schedule for the specified sensor.

**Generic operation: GetUIDs**

The GetUIDs operation returns a list of sensor objects managed by the responding sensor manager.

GetUIDs()

=> (UIDs: array of EUID;

Names: array of ASC;

Types: array of SENSORTYPE)

**UIDs**

is an array of the sensor UIDs.

**Names**

is a corresponding list of the sensor names.

**Types**

is a corresponding list of the sensor types.

**Operation: Edit**

The Edit operation allows an operator to alter the description of an existing sensor.

Edit(Changes: SENSORVARIABLE)

=> ()

**Changes**

is the revised sensor description.

*Sensor Operation***Operation: OperateSensor**

The OperateSensor operation starts a sensor beginning with the specified mission.

OperateSensor(Mission: U16I)

=> ()

**Mission**

is the first mission to be performed.

**Operation: GetDetections**

The GetDetections operation retrieves targets detected by the sensor.

GetDetections()

=> (Detections: array of DETECTIONDATA)

**Detections**

are the sensed targets.

**Operation: SetDetectionThreshold**

The SetDetectionThreshold operation alters the sensor's threshold which is used to discern detections from ambient noise.

SetDetectionThreshold(Threshold: U16I)

=> ()

**Threshold**

is the new threshold to use.

*Canonical Types***Cantype: FRMDATA**

The FRMDATA cantype stores a 2-D image of surveillance data.

FRMDATA: record

time: EDATE;

threshold: S16I;

frame: array of DETECTIONDATA;

end;

**time**

is the time at which the sensor performed the observation.

**threshold**

is the threshold applied to this frame.

**frame**

is the perceived image.

**Cantype: MISSIONDATA**

A MISSIONDATA cantype is used to store a mission's detection data.

**MISSIONDATA: record**

name: ASC;  
type: SENSORTYPE;  
resolution: S16I;  
mission: FRMDEF;  
detectfrms: array of FRMDATA;  
end;

**name**

is the name of the sensor.

**type**

is the kind of sensor.

**resolution**

is the sensor's resolution in deciseconds.

**mission**

is the mission schedule corresponding to the detection data.

**detectfrms**

is the time-series of 2-D frames of detection data that constitute the mission.

**Cronus type:** CT\_Target\_Simulator (subtype of CT\_C2Internet\_Object)

The Target Simulation Manager implements operations that manage a simulated target movement scenario. Each CT\_Target\_Simulator object represents a simulated target. Each target consists of a name, a type, and a sequence of track segments. Each of these track segments is linear and begins at the end of the previous segment.

#### *Target Manipulation Operations*

#### **Generic operation: Create**

The Create operation creates a new simulated target.

```
Create(Name: ASC;  
      Type: TARGETTYPE;  
      StartTime: EDATE;  
      Location: LOC3D)  
=> (ObjectUID: EUID)
```

##### **Name**

is the name of the simulated target.

##### **Type**

is the kind of target that will be simulated.

##### **StartTime**

is the simulated time at which the target will first appear.

##### **Location**

is the initial location of the target.

##### **ObjectUID**

is the UID of the created target.

#### **Operation: MoveTo**

The MoveTo operation adds a track segment to the target trajectory of an existing simulated target. The segment begins at the end location and time of the last previously defined segment.

```
MoveTo(EndLocation: LOC3D;  
      EndTime: EDATE)  
=> ()
```

##### **EndLocation**

is the desired location of the target at the end of the segment added.

##### **EndTime**

is the time at which the target will reach this location.

*Target Retrieval Operations***Generic operation: TargetsInRegion**

The TargetsInRegion operation returns the track segments for those targets present in a specified rectangular region during a particular time interval.

**TargetsInRegion**(StartTime: EDATE;  
                  EndTime: EDATE;  
                  Region: REGION)  
=> (Segments: array of SEGMENT;  
     TargetUIDS: array of EUID;  
     TargetTypes: array of TARGETTYPE)

**StartTime**

is the beginning of the time interval for which targets are requested.

**EndTime**

is the end of the time interval for which targets are requested.

**Region**

is the geographic area in which targets are requested.

**Segments**

are the track segments.

**TargetUIDS**

are the UUIDs of the targets corresponding to the track segments.

**TargetTypes**

are the target types of the targets corresponding to the track segments.

**Cronus type:** CT\_Timer (subtype of CT\_Replicated\_Object)

The timer manager maintains a globally synchronized time of day clock and a collection of experiment timers. The global standard time is periodically broadcast from a selected master clock manager to all other managers, each of which then calculates the difference between the time of their particular processor clock and the global time of day time. Thereafter, the managers estimate the global standard time by adding this adjustment to the time of their local processor clock. We do not correct for the delay incurred while constructing the broadcast message, broadcasting the message to the timer managers or unbundling the message; we assume this delay is small enough not to cause errors in the time calculations. If we require more precise samples, or the time delays prove to be more than we anticipated, the algorithm can be improved.

The manager also supports a collection of replicated timer objects. Each timer object operates as a stopwatch and has an adjustable rate. So, the client of a time object may set the current value and may choose to have it run at twice real time during a demonstration. The timer may be started and stopped at arbitrary instants.

**Implementation:** Time on each timer elapses along a series of time segments. The inflection points of these segments occur whenever the rate changes, of which starting and stopping the clock are particular cases, and when the timer value is explicitly changed. The timer object records a global and timer reference time for the most recent inflection point and estimates the current time from these values using the following formula:

$$\text{TimerValue} = \text{TimerTimeRef} + \text{ElapseRate} * (\text{GlobalTime} - \text{GlobalTimerRef})$$

The GlobalTime value is calculated using the broadcast algorithm described above.

**Generic operation:** GetHostTime

This returns the current value of the host clock.

GetHostTime()

=> (HostTime: EDATE)

HostTime

Processor clock time for this particular manager

**Generic operation:** GetTime

This returns the current global standard time.

GetTime()

=> (Time: EDATE)

Time

Global standard time value

**Generic operation:** SetTime

This allows a client to change the global date, as when a misguided manager changes the date to an incorrect value. The new value will be propagated throughout the collection of managers. This operation does not effect the ID of the master clock. If the new global time is omitted, the receiving host's host clock will be used to get the new date.

This operation requires generic operator rights.

SetTime([Time: EDATE])  
=> ()

**Time**

New global standard time value

#### Generic operation: SelectMasterClock

Invoking this operation nominates the manager receiving the request for the role of master clock. The manager will then determine if another master clock exists and, if so, demote any such managers from the master clock status. Thereafter, every <interval> seconds, the newly elected master clock will synchronize the global times of all other manager.

This operation requires generic operator rights.

SelectMasterClock([Interval: U16I])  
=> ()

**Interval**

New synchronization interval for global standard time, in seconds

#### Generic operation: DeselectMasterClock

When sent to the master clock, causes it to no longer be the master clock.

This operation requires generic operator rights.

DeselectMasterClock()  
=> ()

#### Generic operation: SetSynchInterval

Used to change the synchronization interval. May be submitted to any manager and it will be forwarded to the master.

This operation requires generic operator rights.

SetSynchInterval(Interval: U16I)  
=> ()

**Interval**

New synchronization interval for global standard time, in seconds

**Generic operation: SetQuality**

Used to set quality assessment of a particular clock.

This operation requires generic operator rights.

SetQuality(Quality: S16I)

=> ()

**Quality**

New quality rating for given manager; low numbers, near zero denote preferred masters

**Generic operation: BidForBecomingMaster**

This is used by a manager, during an election, who wants to become the master clock. The manager receiving this request compares his own credentials to those in the bid and invokes a corresponding bid on the sender if the sender loses the bid. Otherwise, it is assumed that the sender will become the master.

BidForBecomingMaster(ElectionBid: BID)

=> ()

**ElectionBid**

Our qualifications for becoming the new master clock

**Generic operation: Synchronize**

This is used, only by the managers, to propagate the global date among the managers. If a master clock manager receives this request from another manager, the receiving manager will assume that the manager issuing the command has been elected master clock manager; the receiving manager will then resign its role as master clock manager. A manager does not process its own invocations of SynchronizeGlobalTime.

Synchronize(Time: EDATE;  
SynchInterval: U16I)

=> ()

**Time**

Current global standard time

**SynchInterval**

Time until we will send the next synch, in seconds.

**Generic operation: SetLoggingLevel**

Change logging parameters of the timer manager.

SetLoggingLevel(LoggingLevel: U16I)

=> ()

**LoggingLevel**

New logging level, by bits: 0x1-synchronization protocol, 0x2-bidding and elections, 0x4-all generic operations, 0x8-all object operations

**Generic operation: ReportStatus**

This operation returns the local date of the host performing the operation, the global date being used by the system and manager specific status information.

**ReportStatus()**

=> (Status: TIMERMGRSTATUS)

**Status**

Status description

**Generic operation: Create**

This function creates a new timer object and returns its UID. If the 'StartRunning' flag is true, the timer will be set to the current time of day and set running in real time (that is, Rate=1). If the 'StartRunning' flag is false, the timer will be set to zero and will await setting by the other timers. The default, when StartRunning is omitted, is true.

This operation requires generic create rights.

**Create([StartRunning: EBOOL];**

[ObjectUID: EUID];

[IACLHints: array of EUID])

=> (ObjectUID: EUID)

**StartRunning**

New timer should be running after creation

**ObjectUID**

Timer should have this particular UID

**IACLHints**

Hints for filling in ACL of new timer object

**ObjectUID**

UID of new timer object

**Operation: GetTime**

Returns the current setting of timer.

**GetTime()**

=> (Time: EDATE)

**Time**

Current simulated time

**Operation: SetTime**

Sets the current time of timer. With no argument, sets sample time to match global timer. If the stop time is set and the timer value is set beyond the stop time, the pending stop command will be cancelled. If the stop time is set and the timer value is set before the stop time, the timer will run until the pending stop time and then stop.

This operation requires operator rights.

SetTime([Time: EDATE])  
=> ()

**Time**

New simulated time setting

**Operation: Reset**

Set timer to value stored in InitialValue field.

This operation requires operator rights.

Reset()  
=> ()

**Operation: SetResetValue**

Sets the value saved in InitialValue for use in Reset operation.

This operation requires operator rights.

SetResetValue(Time: EDATE)  
=> ()

**Time**

New reset value for restarting timer

**Operation: SetRate**

Sets the rate associated with timer.

This operation requires operator rights.

SetRate([RateMultiplier: U16I];  
[RateDivisor: U16I])  
=> ()

**RateMultiplier**

Numerator of elapsed time rate

**RateDivisor**

Demoninator of elapsed time rate

**Operation: Start**

Starts the timer running from its current value. If the StopTime is specified, the timer will run until the specified time and then stop. This command cancels any pending stop request.

This operation requires operator rights.

Start([StopTime: EDATE])  
=> ()

**StopTime**  
Timer will run until it reaches this time

**Operation: Stop**

Stops the timer. Its value will not be reset.

This operation requires operator rights.

Stop()  
=> ()

**Operation: SetStopTime**

The timer will run until the specified time and then stop. If the specified time has already passed, the timer will be stopped immediately at its current setting.

This operation requires operator rights.

SetStopTime(StopTime: EDATE)  
=> ()

**StopTime**  
Timer will run stop when reaches this time

**Operation: SetStepIncrement**

Sets the default value for advancing the timer in steps.

This operation requires operator rights.

SetStepIncrement(StepIncrement: EDATE)  
=> ()

**StepIncrement**  
Timer will advance this many seconds on next step

**Operation: Step**

The timer is started running until count steps have elapsed. If specified, StepIncrement becomes the new default.

This operation requires operator rights.

```
Step([Count: U16I];
     [StepIncrement: EDATE])
=> ()
```

**Count**

Number of steps to allow time to advance

**StepIncrement**

Increment to use for each step; this becomes the new default step increment

**Operation: ReportStatus**

This operation describes the status and parameters of a particular timer object. It returns the current simulated and true time, and the object parameters describing whether the timer is running or not, what rate the timer is set to run at, and whether the timer is set to automatically stop.

```
ReportStatus()
=> (Time: EDATE;
    GlobalTime: EDATE;
    Attributes: TIMEROBJECT)
```

**Time**

Current simulated time value

**GlobalTime**

Current global standard time value

**Attributes**

Timer object instance variables

**Operation: SetLoggingLevel**

Change logging parameters of a particular object.

```
SetLoggingLevel(LoggingLevel: U16I)
=> ()
```

**LoggingLevel**

New logging level; non-zero enables logging

**Canonical Types**

**Cantype: BID**

Bids are broadcast by each candidate manager running for election. If a candidate receives a counter-bid, better than its own, it will withdraw from the election. This competition will eventually leave only one candidate, which will then become the master clock.

**BID: record**

```

SelectedByUser: EBOOL;
IsMaster: EBOOL;
Quality: S16I;
SynchGlobalTime: EDATE;
SynchInterval: U16I;
end;
```

**SelectedByUser**

user elected this manager for new master. Wins if only one has been so selected.

**IsMaster**

Manager won last election. Wins if no user selected a different one.

**Quality**

Quality assessment of host processor clock, usually set by operator. Managers with better clocks have higher quality values.

**SynchGlobalTime**

Slave clocks: global time synch command last was received. Most recently synchronized slave manager will win the bid.

**SynchInterval**

Synchronization period that will be used if we win the bid.

**Cantype: TIMERMGRSTATUS**

A Timer Manager Status structure is kept by each manager in the system. The contents of the structure can be retrieved by invoking the "ReportStatus" operation on the generic timer object.

**TIMERMGRSTATUS: record**

```

HostTime: EDATE;
GlobalTime: EDATE;
Adjustment: EDATE;
AdjAvg: S32I;
AdjDev: S32I;
AdjSamples: S32I;
LoggingLevel: U16I;
Candidate: EBOOL;
Bid: BID;
MasterIsKnown: EBOOL;
MasterHostID: EHOST;
SynchInterval: U16I;
TimeUntilSynch: U16I;
end;
```

**HostTime**

Host processor time for this particular manager.

**GlobalTime**

Global standard time of day

**Adjustment**

Difference between local and global timer values.

**AdjAvg**

Average adjustment value.

**AdjDev**

Std dev of adjustment value in hundredths

**AdjSamples**

Number of adjustment settings used in avg and std dev

**LoggingLevel**

Generic logging level. Each timer may have an individual level too.

**Candidate**

Manager is a candidate for becoming master clock and hasn't lost yet.

**Bid**

Bid submitted for election

**MasterIsKnown**

Set when we know who the master clock is, ie. after we become master or another clock sends a synchronize to us.

**MasterHostID**

For slave clocks. host that invoked last synchronize command.

**SynchInterval**

Number of seconds between synchronizations of global standard time clocks.

**TimeUntilSynch**

Time until next automatic synchronization will be sent.

**Cantype: TIMEROBJECT**

The parameters (instance variables) for each timer object are collected into this "Timer Object" record. The parameters distinguish each timer object, indicated when the timer was last set, whether it is running, what rate it is running at, etc. The values of these parameters can be modified using the timer operations described elsewhere.

**TIMEROBJECT: record**

TimerTimeRef: EDATE;  
GlobalTimeRef: EDATE;  
InitialValue: EDATE;  
StepIncrement: EDATE;

Running: EBOOL;  
AwaitingStop: EBOOL;  
StopTime: EDATE;  
RateMultiplier: U16I;  
RateDivisor: U16I;  
LoggingLevel: U16I;  
end;

**TimerTimeRef**

Timer time when GlobalTimeRef was last sampled.

**GlobalTimeRef**

Global time sampled when current timer segment began

**InitialValue**

Initial setting. The reset operation sets time to this.

**StepIncrement**

Period to use when stepping time in regular increments.

**Running**

True if timer is running

**AwaitingStop**

True if stop is pending

**StopTime**

Time when pending stop command should take effect

**RateMultiplier**

Number of experiment seconds that elapse every real second

**RateDivisor**

Number real seconds for every experiment second

**LoggingLevel**

Logging level for this object

**Cronus type:** CT\_Target\_Report (subtype of CT\_C2Internet\_Object)

The CT\_Target\_Report type describes an object that has been detected by one of the sensing systems. Each object may have additional data tagged to it that describes its attributes, such as anticipated destination or priority.

**Generic operation:** GetTargetDesc

The GetTargetDesc operation returns to the invoker a list of target descriptions for those targets that have been detected within the specified area during the specified time window.

```
GetTargetDesc(Region: REGION;  
              [StartTime: EDATE];  
              [EndTime: EDATE])  
=> (TargetDesc: array of TARGETDESC)
```

**Region**

is the rectangular area within which reported targets are of interest.

**StartTime**

is the beginning of the time window of interest.

**EndTime**

is the end of the time window of interest.

**TargetDesc**

is the list of selected target descriptions.

**Operation:** GetTargetReport

The GetTargetReport operation retrieves the information stored in a Target Report object.

```
GetTargetReport()  
=> (Target: TARGETREPORT)
```

**Target**

is the information retrieved.

**Generic operation:** StoreTargetReport

The StoreTargetReport operation creates a Target Report object, and stores the data provided.

```
StoreTargetReport(TargetReport: TARGETREPORT)  
=> (ReportUID: EUID)
```

**TargetReport**

is the information to be stored.

**ReportUID**

is the unique identifier of the newly created object.

*Canonical Types***Cantype: TARGETDESC**

The TARGETDESC cantype briefly summarizes the information known about a detected target.

**TARGETDESC: record**

```
ReportUID: EUID;  
TargUID: EUID;  
Name: ASC;  
Type: ENUM;  
Destination: LOC3D;  
Priority: S32I;  
Weapon: ASC;  
Region: REGION;  
StartTime: EDATE;  
EndTime: EDATE;  
end;
```

**ReportUID**

is the identifier assigned to the report.

**TargUID**

is the identifier of the simulated target that is associated with this report.

**Name**

is a user-defined name associated with the target.

**Type**

is the kind of target.

**Destination**

is the known or anticipated destination of the target.

**Priority**

is the relative importance of the target.

**Weapon**

is the weapon assignment for the target.

**Region**

is the bounding rectangle of the area in which the target has been detected.

**StartTime**

is the time at which the target was first detected.

**EndTime**

is the time at which the target was last detected.

**Cantype: DETECTIONS**

The DETECTIONS cantype describes a snapshot in time of a single target detection.

DETECTIONS: record

SampleTime: EDATE;  
Location: LOC3D;  
snr: U32I;  
end;

**SampleTime**

is the time at which the target was sensed.

**Location**

is the target's position.

**snr**

is the signal-to-noise ratio of the detection. This value indicates the confidence of the sensor that a target was indeed present at the time and position specified.

**Cantype: SENSORDATA**

The SENSORDATA cantype is a collection of target detections that have been reported by a sensor.

SENSORDATA: record

Sensor: EUID;  
SensorName: ASC;  
Detections: array of DETECTIONS;  
end;

**Sensor**

is the unique identifier of the sensor providing the data.

**SensorName**

is the name of the sensor providing the data.

**Detections**

is the collection of target detections.

**Cantype: TARGETREPORT**

The TARGETREPORT cantype contains a complete description of the status of a detected target.

TARGETREPORT: record

TDesc: TARGETDESC;  
Sensors: array of SENSORDATA;  
end;

**TDesc**

is the summary of the target description.

**Sensors**

is the sensor data for the target.

**Cronus type:** CT\_Vehicle (subtype of CT\_Object)

The CT\_Vehicle type stores information about the operational capabilities of various vehicles (e.g, their maximum speed or range). At the present time, the orientation is toward aircraft.

**Generic operation:** Create

The Create operation defines the capabilities of a new aircraft. When information for a new aircraft is entered, the name of the aircraft is cataloged in the Cronus directory :user:c2inet:vehicles.

This operation requires generic create rights.

```
Create(Plane: PLANEDESC;  
      [ObjectUID: EUID];  
      [IACLHints: array of EUID])  
=> (ObjectUID: EUID)
```

**Plane**

is the description of the new aircraft.

**ObjectUID**

is an optional specification of the UID to be assigned to the newly created object.

**IACLHints**

provides optional hints for initializing the Access Control List for the new object. A detailed description may be found in the annotations for the Create operation on type CT\_Object.

**ObjectUID**

is the UID of the newly created object.

**Generic operation:** ListVehicles

The ListVehicles operation returns a list of all vehicles for which capability data is available.

This operation requires generic listvehicles rights.

```
ListVehicles()  
=> (VehicleNameList: array of ASC)
```

**VehicleNameList**

is the list of vehicles for which data is available.

**Generic operation:** ListVehiclesWithCapability

The ListVehiclesWithCapability operation looks up all of the available vehicles and returns a list of those matching or surpassing the specified capabilities.

This operation requires generic listvehicles, capabilitysearch rights.

```
ListVehiclesWithCapability([VehicleName: ASC];  
    [VehicleType: ASC];  
    [MaximumSpeed: U16I];  
    [MaximumWeight: U16I];  
    [MaximumRange: U16I];  
    [MaximumFuel: U16I];  
    [TakeoffLength: U16I];  
    [LandingLength: U16I])  
=> (VehicleNameList: array of ASC)
```

**VehicleName**

is a character string that matches all or part of the vehicle name.

**VehicleType**

is a character string that matches all or part of the vehicle type.

**MaximumSpeed**

is the required aircraft speed (in kilometers per hour).

**MaximumWeight**

is the required aircraft weight (in kilograms).

**MaximumRange**

is the required aircraft range (in kilometers).

**MaximumFuel**

is the required aircraft fuel capacity (in liters).

**TakeoffLength**

is the required aircraft takeoff distance (in meters).

**LandingLength**

is the required aircraft landing distance (in meters).

**VehicleNameList**

is the list of the vehicles meeting the specified requirements.

*Canonical Types***Cantype: VEHICLEDESC**

The VEHICLEDESC cantype is used to represent the characteristics of a vehicle.

**VEHICLEDESC: record**

```
VehicleName: ASC;  
VehicleType: ASC;  
MaximumSpeed: U16I;  
MaximumWeight: U32I;  
MaximumRange: U16I;  
MaximumFuel: U32I;  
end;
```

**VehicleName**

is the name of a vehicle, e.g. F16.

**VehicleType**

is the vehicle class, e.g., attack aircraft, light truck, etc.

**MaximumSpeed**

is the maximum vehicle speed in kilometers per hour.

**MaximumWeight**

is the maximum vehicle weight in kilograms.

**MaximumRange**

is the maximum vehicle range in kilometers without refueling.

**MaximumFuel**

is the maximum fuel capacity of the vehicle in liters.

**Cantype: PLANEDESC**

The PLANEDESC cantype is used to represent the characteristics of an aircraft.

**PLANEDESC: record**

```
VehicleInfo: VEHICLEDESC;  
TakeoffLength: U16I;  
LandingLength: U16I;  
end;
```

**VehicleInfo**

is a description of the vehicle attributes.

**TakeoffLength**

is the minimum takeoff distance in meters.

**LandingLength**

is the minimum landing distance in meters.

**Cronus type:** CT\_Weather\_Forecast (subtype of CT\_Weather\_Data)

A CT\_Weather\_Forecast is a summary of the weather forecast at a given location for a given time window.

**Generic operation:** Create

Allow the user to create a forecast for a specified time period.

This operation requires generic create rights.

```
Create([ObjectUID: EUID];  
       [IACLHints: array of EUID];  
       WeatherForecast: WXCASTDATA)  
=> (ObjectUID: EUID)
```

**ObjectUID**

is an optional specification of the UID to be assigned to the newly created object.

**IACLHints**

provides optional hints for initializing the Access Control List for the new object. A detailed description may be found in the annotations for the Create operation on type CT\_Object.

**WeatherForecast**

is the weather being forecast.

**ObjectUID**

is the UID of the forecast created.

**Generic operation:** ShowForecastNearLocation

The ShowForecastNearLocation operation returns the weather forecast for the requested forecast date and the location nearest to the one specified.

This operation requires generic listforecasts rights.

```
ShowForecastNearLocation(RequestedForecastDate: EDATE;  
                          Location: LOC)  
=> (WeatherForecast: WXCASTDATA)
```

**RequestedForecastDate**

is the date and time for which the forecast is wanted.

**Location**

is the place for which the forecast is wanted.

**WeatherForecast**

is the requested weather forecast.

**Generic operation: DeleteOutdatedForecasts**

The DeleteOutdatedForecasts operation deletes weather forecasts that have outlived their usefulness. A forecast is considered to have expired if the current date is later than the end of the forecast validity period.

This operation requires generic deleteoutdated rights.

DeleteOutdatedForecasts()

=> (NumberOfDeletedForecasts: U16I)

**NumberOfDeletedForecasts**

is the number of forecasts successfully deleted by the operation.

**Canonical Types**

**Cantype: WXCASTDATA**

The WXCASTDATA cantype summarizes the relevant weather parameters for a weather forecast.

WXCASTDATA: record

ForecastDate: EDATE;  
BeginValidityPeriod: EDATE;  
EndValidityPeriod: EDATE;  
ForecastData: WXINFO;  
end;

**ForecastDate**

The date and time when the forecast was formulated.

**BeginValidityPeriod**

The beginning of the period for which the forecast is valid.

**EndValidityPeriod**

The end of the period for which the forecast is valid.

**ForecastData**

is the weather forecast data.

**Cronus type:** CT\_Weather\_Data (subtype of CT\_C2Internet\_Object)

CT\_Weather\_Data defines meteorological information for a given geographical location. The information that is defined is that which is common to both weather reporting and forecasting. No operations are defined for CT\_Weather\_Data.

#### *Canonical Types*

**Cantype:** CLOUDCOVERAGE

The CLOUDCOVERAGE cantype describes the relative density of the cloud cover.

CLOUDCOVERAGE: { NONE, LIGHT, HEAVY };

**NONE**

indicates no clouds.

**LIGHT**

indicates light clouds.

**HEAVY**

indicates heavy clouds.

**Cantype:** CLOUDTYPE

The CLOUDTYPE cantype describes a particular kind of cloud formation.

CLOUDTYPE: { CIRRUS, NIMBUS, CUMULUS, STRATUS, CIRROCUMULUS, CIRROSTRATUS, NIMBOSTRATUS, CUMULONIMBUS, STRATOCUMULUS, ALTOCUMULUS, ALTOSTRATUS };

**CIRRUS**

is a wispy white cloud usually of minute ice crystals formed at altitudes of 6000 to 12000 meters.

**NIMBUS**

is a rain cloud that is of uniform grayness and extends over the entire sky.

**CUMULUS**

is a massy cloud form having a flat base and rounded outlines often piled up like a mountain.

**STRATUS**

is a cloud form of greater horizontal extension and comparatively lower altitude (1000 to 2000 meters) than the cumulostratus or cirrostratus.

**CIRROCUMULUS**

is a cloud form of small white rounded masses at a high altitude usually in regular groupings forming a mackerel sky.

**CIRROSTRATUS**

is a fairly uniform layer of high stratus clouds that are darker than cirrus clouds.

**NIMBOSTRATUS**

is a low dark gray rainy cloud layer.

**CUMULONIMBUS**

is a cumulus cloud spread out in the shape of an anvil extending to great heights.

**STRATOCUMULUS**

is a stratified cumulus cloud consisting of large balls or rolls of dark cloud which often cover the whole sky, especially in winter.

**ALTOCUMULUS**

is a fleecy cloud formation consisting of large whitish globular cloudlets with shaded portions.

**ALTOSTRATUS**

is a cloud formation similar to cirrostratus but darker and at a lower level.

**Cantype: WEATHERTYPE**

The WEATHERTYPE cantype summarizes the weather conditions for a given location.

WEATHERTYPE: { CLEAR, CLOUDS, FOG, RAIN, SNOW };

**CLEAR**

indicates clear weather.

**CLOUDS**

indicates overcast conditions.

**FOG**

indicates fog.

**RAIN**

indicates precipitation.

**SNOW**

indicates precipitation.

**Cantype: WXINFO**

The WXINFO cantype summarizes the relevant weather parameters.

**WXINFO: record**

Location: LOC;

Ceiling: U16I;

CloudCoverage: CLOUDCOVERAGE;

CloudTypes: array of CLOUDTYPE;

SurfaceVisibility: U32I;

VisibilityRestrictions: ASC;  
Weather: WEATHERTYPE;  
BarometricPressure: U16I;  
Temperature: S16I;  
DewPoint: S16I;  
SurfaceWindDirection: U16I;  
AvgSurfaceWindSpeed: U16I;  
MaxSurfaceWindSpeed: U16I;  
Remarks: ASC;  
Station: ASC;  
end;

**Location**

is the site for which the weather is specified.

**Ceiling**

in meters.

**CloudCoverage**

is the extent of cloud coverage.

**CloudTypes**

are the kinds of clouds present.

**SurfaceVisibility**

in meters.

**VisibilityRestrictions**

describes any abnormal conditions impeding visibility.

**Weather**

is a summary of the weather condition.

**BarometricPressure**

in millibars.

**Temperature**

in degrees Centigrade.

**DewPoint**

in degrees Centigrade.

**SurfaceWindDirection**

in degrees, 0 - 360.

**AvgSurfaceWindSpeed**

in kilometers per hour.

**MaxSurfaceWindSpeed**

in kilometers per hour.

**Remarks**

are miscellaneous comments provided by the meteorologist.

**Station**

is the name of the responsible weather station.

**Cronus type:** CT\_Weather\_Report (subtype of CT\_Weather\_Data)

A CT\_Weather\_Report is a summary of the weather reported at a given location and time.

**Generic operation:** Create

Report the weather conditions for a given location. The report provided will replace any existing previously provided report for the given location. Hence performing a create operation requires (non-generic) remove permission.

This operation requires generic create rights.

```
Create([ObjectUID: EUID];  
       [IACLHints: array of EUID];  
       WeatherReport: WXREPORTDATA)  
=> (ObjectUID: EUID)
```

**ObjectUID**

is an optional specification of the UID to be assigned to the newly created object.

**IACLHints**

provides optional hints for initializing the Access Control List for the new object. A detailed description may be found in the annotations for the Create operation on type CT\_Object.

**WeatherReport**

is the weather being reported.

**ObjectUID**

is the UID of the report created.

**Generic operation:** ShowWeatherNearLocation

The ShowWeatherNearLocation operation returns the most up-to-date weather report for the location nearest to the one specified.

This operation requires generic listreports rights.

```
ShowWeatherNearLocation(Location: LOC)  
=> (WeatherReport: WXREPORTDATA)
```

**Location**

is the location for which the reported weather is desired.

**WeatherReport**

is the requested weather report.

*Canonical Types*

**Cantype: WXREPORTDATA**

The WXREPORTDATA cantype summarizes the relevant weather parameters for a weather report.

WXREPORTDATA: record

ReportDate: EDATE;  
ReportData: WXINFO;  
end;

**ReportDate**

is the date and time at which the report was formulated.

**ReportData**

is the weather report data.